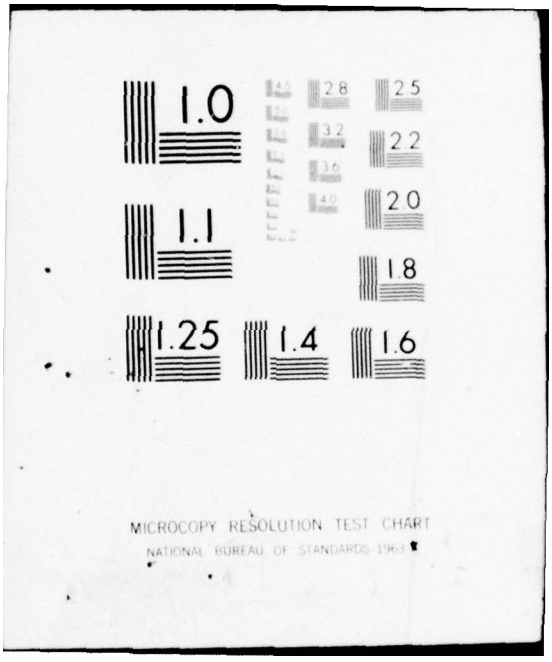
AD-A033 296 BDM CORP EL PASO TEX F/G 9/2
ON THE DESIGN OF AN EXECUTIVE PROGRAM AND A COMMON SIMULATION L--ETC(U)
SEP 76 A F MALMBERG F29601-74-C-0017
UNCLASSIFIED BDM/E-47-75-F-0017 AFWL-TR-75-272 NL

1 OF 3
AD
A033296


1.0
1.1
1.25
1.4
1.6
2.8
2.5
3.2
2.2
3.6
2.0
4.0
1.8
MICROCOPY RESOLUTION TEST CHART
NATIONAL BUREAU OF STANDARDS-1963

AFWL-TR-75-272

AFWL-TR-75-272

ON THE DESIGN OF AN EXECUTIVE PROGRAM AND A COMMON SIMULATION LANGUAGE TRANSLATOR FOR A SYSTEMS ANALYSIS PROGRAM

BDM Corporation
El Paso, Texas 79905

September 1976

Final Report

Approved for public release; distribution unlimited.

This research was sponsored by the Defense Nuclear Agency under Subtask Z99QAXTC022, Work Unit 52, Work Unit Title: Interface Program for Circuit Analysis.

Prepared for
Director
DEFENSE NUCLEAR AGENCY
Washington, DC 20305

AIR FORCE WEAPONS LABORATORY
Air Force Systems Command
Kirtland Air Force Base, NM 87117

This final report was prepared by the BDM Corporation, El Paso, Texas, under Contract F29601-74-C-0017, Job Order WDNE1308, with the Air Force Weapons Laboratory, Kirtland Air Force Base, New Mexico. Mr. White (ELP) was the Laboratory Project Officer-in-Charge.

When US Government drawings, specifications, or other data are used for any purpose other than a definitely related Government procurement operation, the Government thereby incurs no responsibility nor any obligation whatsoever, and the fact that the Government may have formulated, furnished, or in any way supplied the said drawings, specifications, or other data is not to be regarded by implication or otherwise as in any manner licensing the holder or any other person or corporation or conveying any rights or permission to manufacture, use, or sell any patented invention that may in any way be related thereto.

This report has been reviewed by the Information Office (OI) and is releasable to the National Technical Information Service (NTIS). At NTIS, it will be available to the general public, including foreign nations.

This technical report has been reviewed and is approved for publication.

A. Brent White

A. BRENT WHITE
Project Officer

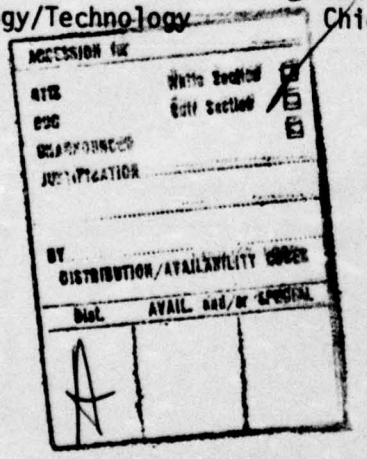
FOR THE COMMANDER

Larry W. Wood

LARRY W. WOOD
Lt Colonel, USAF
Chief, Phenomenology/Technology Branch

James L. Griggs Jr.

JAMES L. GRIGGS, JR.
Colonel, USAF
Chief, Electronics Division



DO NOT RETURN THIS COPY. RETAIN OR DESTROY.

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

REPORT DOCUMENTATION PAGE	READ INSTRUCTIONS BEFORE COMPLETING FORM
1. REPORT NUMBER: AFWL-TR-75-272	2. GOVT ACCESSION NO. / 3. RECIPIENT'S CATALOG NUMBER
4. TITLE (and Subtitle): ON THE DESIGN OF AN EXECUTIVE PROGRAM AND A COMMON SIMULATION LANGUAGE TRANSLATOR FOR A SYSTEMS ANALYSIS PROGRAM	5. TYPE OF REPORT & PERIOD COVERED: Final Report
	6. PERFORMING ORG. REPORT NUMBER: BDM/E-47-75-F-0017
7. AUTHOR(s): Allan F. Malmberg	8. CONTRACT OR GRANT NUMBER(s): F29601-74-C-0017
9. PERFORMING ORGANIZATION NAME AND ADDRESS: BDM Corporation, 6070 Gateway East Blvd., Suite 411, El Paso, TX 79905	10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS: 62704H, WDNE1308
11. CONTROLLING OFFICE NAME AND ADDRESS: Director, Defense Nuclear Agency, Washington, D.C. 20305	12. REPORT DATE: September 1976
	13. NUMBER OF PAGES: 236
14. MONITORING AGENCY NAME & ADDRESS (if different from Controlling Office): Air Force Weapons Laboratory (ELP), Kirtland AFB, NM 87117	15. SECURITY CLASS. (of this report): UNCLASSIFIED
	15a. DECLASSIFICATION/DOWNGRADING SCHEDULE

16. DISTRIBUTION STATEMENT (of this Report)

Approved for public release; distribution unlimited.

17. DISTRIBUTION STATEMENT (of the abstract entered in Block 20, if different from Report)

18. SUPPLEMENTARY NOTES

This research was sponsored by the Defense Nuclear Agency under Subtask Z99QAXTC022, Work Unit 52, Work Unit Title: Interface Program for Circuit Analysis.

19. KEY WORDS (Continue on reverse side if necessary and identify by block number)

Analysis
CADA
Data Base Systems
Computer Aided Analysis
Data Management
Network Simulation
Executive Computer Programs

20. ABSTRACT (Continue on reverse side if necessary and identify by block number)

Development work towards the realization of an executive program and language translator for use with a systems analysis program for vulnerability assessment is described. The executive program permits sequencing of arbitrary application modules on a stand alone basis and provides for exchange of data between modules. An Executive Control Language is described for programming the actions of the executive program. The Common Simulation Language is described which incorporates many of the features of the NET-2, SCEPTRE, and CIRCUS-2 simulation languages. Techniques and difficulties associated with the automated translation of Common

over

DD FORM 1 JAN 73 1473 EDITION OF 1 NOV 65 IS OBSOLETE

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

391886

(Blk 20)

Simulation Language text to the equivalent NET-2, SCEPTRE, and Circus-2 language texts are discussed. Programs for automatically generating LR(k) parsers and multi-scanner lexical analyzers are also described.

CONTENTS

ILLUSTRATIONS

SECTION I

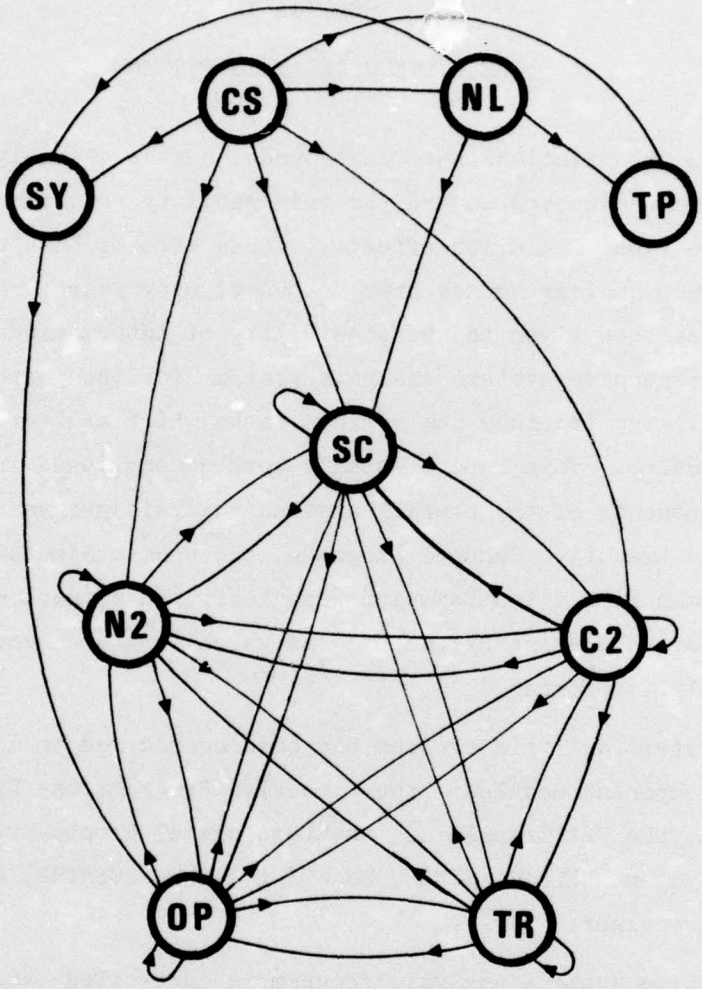
THE SYSTEMS ANALYSIS PROGRAM

Various organizations have developed a number of analysis modules which have been directed toward the vulnerability assessment of weapons systems to nuclear radiation effects. These efforts have been funded by the Defense Nuclear Agency over a span of many years. Recently, attention has been given to the possibility of interfacing these modules into a comprehensive systems analysis program for the purpose of automatically linking together the various tasks which are performed by the separate modules. This report details work accomplished in realizing certain components of the overall systems analysis program: the Executive Program, the Executive Control Language, the Common Simulation Language, and the Common Simulation Language Translator. A companion report discusses the Data Base Management System program which is also a component of the systems analysis program.

The systems analysis program has been considered to include the following component modules: the Executive Program, the Data Base Management System, the CSL Translator, the Topological Processor, the SUPERSAP data base system, NELSIM, SYNAP, TRAFFIC, NET-2, SCEPTRE, CIRCUS-2, and an output processor.

The entire systems analysis program is controlled through the Executive Control Language. Individual modules are sequenced by the Executive Program. The Data Base Management System is used as a repository for input and output data files, problem descriptions, parameter data, reports, data bases, etc.

The number of ways in which the modules can interact is very large. Figure 1 attempts to show this interaction as a state diagram; however, entry and exit from the state diagram can occur at any node, and the Executive Program and Data Base Management System can interact along any path in the diagram. SUPERSAP has not been included in the diagram.



Legend:
- CS - CSL Translator
- C2 - CIRCUS-2
- NL - NELSIM
- N2 - NET-2
- OP - Output Processor
- SC - SCEPTRE
- SY - SYNAP
- TP - Topological Processor
- TR - TRAFFIC

Figure 1. Interactions of Systems Analysis Program Modules

The existing modules have been developed over a period of time with no regard for intermodule compatibility. Thus, the data formats and file organizations which are used at the input and output interfaces for each module are generally incompatible. It is almost always necessary to perform a transformation on the data output from a module so that it may become acceptable input to another module. This transformation is not easily accomplished. One approach is to rewrite the existing modules using a standardized data format. Another approach involves storing the data format as part of the data itself, with a data reformatting language used to program a data translator which then accomplishes the required conversion. A third approach involves the writing of special data conversion programs for each permissible module-module interface; ignoring the interaction of the Executive and Data Base Management System modules it can be seen from Figure 1 that a minimum of 35 translation programs are required. Thus, data compatibility is a serious problem which must be solved before the projected systems analysis program can be realized. It has not been feasible to solve this problem during the course of the investigation.

SECTION II

EXECUTIVE PROGRAM DESIGN

1. THE EXECUTIVE PROGRAM

The Executive Program is responsible for the control of the entire systems analysis calculation. It calls the various application modules in the desired sequence and is responsible for handling files and data required as input or generated as output by the application modules. The actions of the Executive Program are controlled by a user supplied input deck, written in the Executive Control Language (ECL).

The Executive Program operates as a stand alone program, exchanging data with application modules through system files. Similarly, each application module operates in a stand alone mode. The Executive Program calls application modules, with control returning to the Executive following completion of execution of the called module. Successful incorporation of a module into the overall structure does not require any modification of the module itself other than that which may be desirable for passing information between the module and the Executive or other modules.

Operation of the entire complex begins by a simple control card call which loads and executes the Executive Program. The Executive then reads input cards which describe the actions to be taken by the Executive using the Executive Control Language. If another module is required to perform a task the Executive generates a sequence of control cards which load and execute that module (including any file handling which might be required) and then recall the Executive. The Executive then saves its current status on an executive storage area file and calls an operating system routine which transfers control from the normal control card sequence to the control card sequence which has just been created by the Executive. The Executive then terminates its execution.

The operating system now begins processing control cards from the auxiliary control card file which was just created by the Executive. These control cards may specify any arbitrary action, including the loading and execution of application modules. The last control card on the auxiliary file always recalls the Executive. The Executive retrieves its status

information from the executive storage area file and continues execution where it had left off, reading additional commands from the ECL input deck.

The above process is repeated as necessary until all steps in the systems analysis problem have been completed. This is signaled by a RETURN statement in the ECL input deck. At this point the Executive calls an operating system routine which reestablishes the original control card file and processing of control cards from that file is resumed at the point where it had been interrupted. Normal control card processing then continues until the end of the job.

From the above description it can be seen that the systems analysis program complex consists of the Executive Program, a number of application modules, files associated with the Executive operation, and other files associated with the application modules. The Executive Program files are 1) the primary control card file, 2) the auxiliary control card file, created by the Executive, 3) the ECL input deck file, 4) a procedure library permitting the user to access ECL and control card procedures, 5) the executive storage area file which saves the status of the Executive when the Executive is out of core, and 6) a system data file which permits exchange of information on a word by word basis between the Executive and the application modules. The entire systems analysis complex is depicted schematically in Figure 2. The Executive Program concept can be implemented on any computer system which has the ability to switch to alternate control card streams. This feature is available on both IBM 360/370 and CDC 6000/7000 series computers.

The auxiliary control card file is created by the executing ECL program. The control card images are embedded within the ECL program and procedures called by that program. This embedding is accomplished through specification of a control card section which contains the control card images. Provisions are available in ECL procedures for dynamically altering character strings on the control cards. Whenever an ECL control card section is encountered during the execution of an ECL program, any required string substitutions are performed, and the entire control card section is transmitted to the auxiliary control card file. The Executive automatically appends an additional control card at the end of the file which recalls the Executive Program. Control is then passed to the auxiliary control card file. There

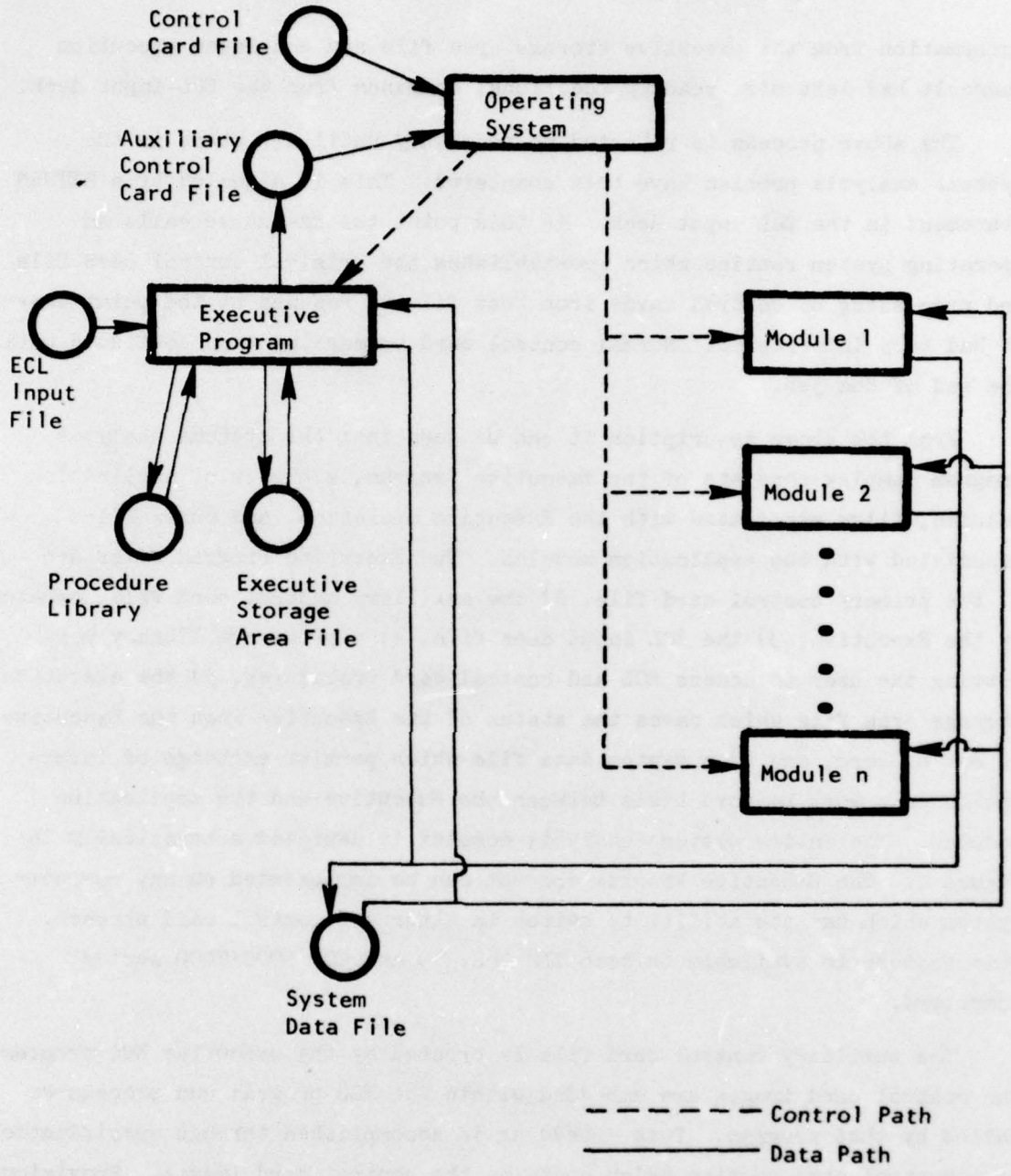


Figure 2. Executive Program Configuration

are no restrictions on the control cards contained within the control card section, thus this control technique is not restricted to particular computers and operating systems.

The procedure library is maintained through the ECL. It is a repository for procedures which may be called by the ECL main program or from another procedure. Both parameter values and character strings may be passed when a procedure is called. Details for procedures are discussed later under the description of ECL.

The executive storage area file is used to store all data associated with a particular execution of the Executive Program when the Executive Program has relinquished control to the auxiliary control card file. During this time the Executive Program is not in core. In order that the Executive be able to resume operation at the point where it was interrupted it must be able to save and retrieve all dynamic data using the executive storage area file. Thus, data are written to this file just prior to transferring control from the Executive to the auxiliary control card file, and data are read back to core when Executive Program operation is resumed, following exhaustion of the auxiliary control card file.

The system data file is a special file which permits the exchange of information on a word by word basis between the Executive Program and the application modules. It can also be used to exchange information directly between application modules. Typical information included on this file would be completion status of application modules, parameter values which must be passed from module to module, etc. Data on this file are stored using the FORTRAN namelist convention so that the order in which data appear is not important. Data are labeled on this file with the data name which permits a free form exchange of information between the various modules.

The ECL permits the user to specify arbitrary names for values and strings generated and manipulated in the ECL program. These value and string variables are stored and accessed through a hash table in the executive storage area, using the variable name as the access data for the hashing mechanism. Thus, it is always possible to efficiently access a variable through its name. This feature works very well with the system data file which contains data labeled by name. Thus, the names which are used in the

ECL program are global names which can be accessed from any part of the ECL program, from procedures called by the program, and, through the system data file, by any application module.

When the Executive Program is initially invoked it must read the entire ECL input deck and store this deck within the executive storage area. This is necessary for several reasons: 1) all procedure library maintenance commands must be processed and the procedure library updated before execution of the ECL program begins, 2) all temporary procedures must be processed, 3) the ECL program and all procedures which are called must be compiled, and 4) the locations of all ECL statement labels must be known prior to execution to permit GO TO statements to be properly handled.

Compilation of the ECL program is done internally by the Executive Program and not by an external compiler. The compiled program is in the form of a table of operation triplets, similar to the internal intermediate form used for representing programs by many existing compilers. However, translation of the triplet table into executing machine code is not done by the Executive Program. Instead, an interpretive execution is employed. Although interpretive execution is slower than direct execution, it is expected that the principal contribution to computer time in a systems analysis computation will be due to the application modules. The effort required to produce a directly executing program does not appear warranted.

Parsing and lexical analysis for the ECL input deck is routinely accomplished using the LR(k) parser and lexical analyzer described elsewhere in this report. The internal triplet table is easily constructed by semantic routines which are coupled with the parsing action.

2. THE EXECUTIVE CONTROL LANGUAGE

The Executive Control Language (ECL) enables the user to program the various steps of the systems analysis problem. The ECL controls the operation of the Executive Program, which, in turn, invokes the application modules which actually perform the various systems analysis tasks.

ECL consists of a subset of the FORTRAN language augmented by special features which are required for controlling the systems analysis program complex. Since the Executive Program invokes application modules through

an alternate control card stream, it is necessary to have a means of handling control card images using ECL. Thus, ECL permits the user to embed control card images within the higher level ECL text stream. Futhermore, it is possible to do character string replacement in the control card images, permitting dynamic generation of control cards.

The ECL supports procedures through a procedure library. Thus, ECL contains maintenance commands for entering, modifying, deleting, and listing the contents of the procedure library. Procedures may also be defined without entry into the library.

The systems analysis complex permits the exchange of data between the Executive Program and the application modules. Facilities are included in ECL for managing data exchange for the Executive Program.

The elements of ECL are described below. The ECL input deck may consist of two parts: a procedure definition and procedure library maintenance part, and the main ECL program. Either part may be omitted; however, if both parts are present, the procedure definition and procedure library maintenance part must come first. The grammar for ECL is given in Appendix N.

a. Card Formatting Conventions

The formatting rules of FORTRAN are used for all cards except those which are associated with control card sections. The FORTRAN formatting conventions are: comments are indicated by a "C" in column 1, card continuation is indicated by a nonzero character in column 6, columns 1-5 are reserved for an integer label field (except on comment cards), and columns 7-72 are used for statement specification. Columns 73-80 may contain arbitrary information.

b. Data Types

Data may be represented in ECL as real numbers or as character strings. Real numbers are written using a string of digits with optional sign and decimal point, and with an optional exponent. The plus sign is optional for positive numbers. The decimal point is not required if it occurs to the right of the last magnitude digit. The exponent, if used,

must always follow immediately after the magnitude and begin with the letter E. The exponent value is expressed as a signed integer, representing a power of ten, with the plus sign optional for positive exponents. Examples of legal number specifications are:

```
25
+2.34
-35.78
4.E+6
49.789E-3
65E7
```

Strings may be composed of any of the keypunch characters, including the blank, and may be from 1 to 80 characters in length. The string is delimited at the beginning and end by the $ symbol. If it is necessary to include the $ symbol as part of the string, it must be written with a $ prefix symbol which acts as an escape symbol. Examples of strings are:

```
$CAT$
$THIS IS A STRING.$
$123$$ABC$
$$$$
$/$
```

Both strings and numbers may be represented symbolically through the use of identifiers. An identifier is any alphameric string, starting with a letter, with a length of six characters or less. Since all numbers are represented as real numbers, there is no integer significance associated with identifiers beginning with the letters I through N as in FORTRAN.

c. Arithmetic Expressions

The user may employ arithmetic expressions, involving the operators +, -, *, /, and **; these operators have their normal FORTRAN meaning. Parentheses may be used for grouping purposes, with nesting permitted to any level. Operands in arithmetic expressions include identifiers and numbers; however, identifiers which represent strings must not be used in arithmetic expressions. The hierarchy of operators in arithmetic evaluation is identical to that of FORTRAN. Since all quantities which enter into arithmetic expressions are real numbers there is no data type conversion involved.

d. Logical Expressions

ECL permits logical expressions to be constructed of any complexity, using numbers, identifiers, arithmetic expressions, relational operators, and logical operators. Parentheses may be used for grouping purposes, with nesting to any level permitted. The relational operators .EQ., .NE., .LE., .LT., .GE., and .GT. are allowed. The logical operators .AND., .OR., and .NOT. are permitted. Logical expressions are constructed in the same manner as in FORTRAN. Identifiers which represent strings must not be used in logical expressions.

e. Replacement Statements

There are two types of replacement statements allowed in ECL. The most general type is of the form A = E, where E is an arithmetic expression and A is any identifier. The other type is of the form A = S, where S is a string or an identifier representing a string, and A is any identifier; A will represent a string following the replacement operation of the second type.

f. Control Statements

Unconditional branches to a specified label are made as follows:

GO TO n

where n is an ECL label.

The logical IF is available for conditional execution of a statement. It has the form:

IF(L) s

where L is a logical expression and s is any ECL statement except another logical IF. If the logical expression is true, then statement s is executed, otherwise statement s is skipped and the next ECL statement is executed.

The CONTINUE statement is available. It is a do-nothing statement and control is always passed to the next statement. It is useful when control must be passed to the start of a control card section; since labels cannot be attached to the start of the control card section, a labeled CONTINUE statement may be used just ahead of the control card section for this purpose.

The RETURN statement is used for return of control from a called procedure back to the calling routine. If a RETURN occurs in the main ECL program, control is passed back to the primary control card file.

g. Procedures

The user may specify procedures using ECL. A procedure may be catalogued in the procedure library or it may be specified as a local procedure, accessible only to the ECL input deck in which it is specified.

A procedure may be defined with or without a list of formal parameters. Formal parameters are used to pass numerical values and strings from the calling program to the procedure. In addition, numerical values may be returned to the calling program using formal parameters. Procedures which utilize formal parameters are written in terms of the formal parameters. Since procedures may call other procedures, a formal parameter may be passed through several levels of procedure calls.

Procedures are defined in the procedure definition and procedure library maintenance part of the deck and must precede the main ECL program. The first card of the procedure definition has one of four forms:

```
DEFINE, procname
DEFINE, procname(parameterlist)
STORE, procname
STORE, procname(parameterlist)
```

where the keyword DEFINE indicates that the procedure is locally defined and is not to be entered into the procedure library, and the keyword STORE indicates that the procedure is globally defined and is to be entered into the procedure library. If a procedure is locally defined and a procedure with the same name already exists in the library, the locally defined procedure will be used by the ECL program. If a procedure is globally defined and a procedure with the same name already exists in the library, the new procedure will replace the old procedure in the library.

The procname field specifies the procedure name. Procedure names are composed of one to six letters. The parameterlist field is the list of formal parameters used in the procedure. These parameters are of two types: value type and string type. Both types may be passed as parameters in a procedure call within the procedure being defined. Formal parameters are separated by commas.

Value type formal parameters are represented by identifiers. String type formal parameters are represented by identifiers delimited by $ symbols. Thus, an example of the first line of a procedure definition is:

```
STORE, ANALYZ(TIME,FREQ,$TAPE7$,$ABC$,ERROR)
```

A procedure may contain two kinds of cards. ECL cards contain statements in the ECL language. The other kind of card is a control card. Value type formal parameters may not be used in connection with control cards; conversely, string type formal parameters may only be used in connection with control cards. If a character string corresponding to a formal parameter of the value type appears on a control card, no action involving the formal parameter will be taken. Similarly, if a character string corresponding to a formal parameter of the string type occurs on an ECL card, no action involving the formal parameter will be taken.

A procedure may be written using only ECL statement types which may appear in the main ECL program. Thus, DEFINE, STORE, DELETE, and LIST statements may not appear within a procedure.

A procedure is referenced using the CALL statement:

```
CALL procname
CALL procname(argumentlist)
```

where procname is the procedure name, and argumentlist is a list of arguments representing values and strings in one-to-one correspondence with the formal parameters in the procedure definition. Arguments are separated by commas. Strings may be represented by an actual string, enclosed in $ symbols, or by an identifier representing a string. Values may be represented by arithmetic expressions, by an identifier representing a value, or by a number. For example:

```
CALL ANALYZ(2.45,AFC/2.5+BIAS,FILE34,$REWIND$,FLAG)
```

In this example (which corresponds to the procedure definition example above) the first, second, and fifth arguments are values, while the third and fourth arguments are strings.

A procedure definition is terminated by:

END procname

where procname is the name of the procedure being defined.

h. Data Communication Statements

The Executive Program utilizes the system data file to communicate data between itself and the application modules. Thus, data may be written onto this file by the Executive Program to be read by a particular application module, and conversely an application module may write data on the file to be read by the Executive Program. Since data is written using the FORTRAN namelist format, it is not necessary that a particular order be observed on the file. The system data file can be regarded as a labeled data base of variable size.

ECL has two statements to permit the user to interact with the system data file. The file is always rewound before either statement is executed. Thus, old information is always lost from the file when an ECL WRITE statement is executed.

The WRITE statement has the format:

WRITE(valuelist)

where valuelist is a list of identifiers whose values are to be written to the file. Identifiers in the list are separated by commas. Since both the identifier name and the identifier value are written to the file, it is seen that numbers, strings, and arithmetic expressions cannot be included in the valuelist field. An example of a WRITE statement is:

WRITE(TIME,FREQ,ERROR)

The system data file may be read by using the READ statement:

READ

There are no other parameters involved with the READ statement since all information is contained on the system data file. If a value is read for an identifier already in the executive storage area, the old value is replaced by the new value. If the identifier is not already resident in the executive storage area, a space is created for it and the value is stored.

i. Control Card Sections

The user may declare control card sections within both the main ECL program and in procedures. This is the means by which control cards are entered into the Executive Program system. In fact, a procedure could contain only control card sections. As many control card sections may be declared as needed.

When the ECL program is executed and a control card section is encountered, any specified string replacements are made to the control card images. Then all control cards in that section are queued onto the auxiliary control card file, control is relinquished by the Executive Program, and the operating system begins processing control cards from the auxiliary file. The last control card on the file is automatically added by the Executive Program when the file is created, and it returns control back to the Executive Program which then resumes ECL processing following the control card section which caused the interruption.

Thus, control card sections may contain arbitrary card images and have provisions for dynamic modification through string type formal parameters whenever they appear in a procedure.

The control card section is introduced by:

/CC/

This is followed by one or more control cards. The section is terminated by:

/ENDCC/

String type formal parameters must appear in the control card images exactly as they appear in the formal parameter list in the procedure definition, i.e., they are identifiers enclosed by $ symbols. Since the identifier may be arbitrary it is easy to choose identifiers which do not conflict with control card constructs of the same form for which string replacement is not intended. When the string replacement occurs, the width of the replaced field agrees with the length of the string in the procedure call (not counting the $ symbols), not with the width of the formal parameter name. Thus, characters on the control card following the replacement operation may be shifted left or right from their original positions.

j. Procedure Library Maintenance

Facilities are available in ECL for maintaining the procedure library. All library maintenance cards must occur as the first cards in the ECL input deck.

A procedure may be stored into the procedure library by using a STORE statement followed by the procedure definition. If a procedure with the same name previously existed in the library it is replaced with the new procedure.

A procedure in the library can be deleted using the DELETE feature:

DELETE, procname

where procname is the procedure name.

A contents of the procedure library can be listed by including the card:

LIST

k. END Card

The ECL input deck is terminated with the card:

END

SECTION III

THE COMMON SIMULATION LANGUAGE

1. GENERAL

a. Character Set

The Common Simulation Language is composed of a character set which is used to construct the keywords, identifiers, delimiters, and other elements of the language. The characters used are the numerical characters 0 through 9, the alphabetic characters A through Z, and the special characters ()+*/.,$- and =. The blank may be used optionally but is not required by the language.

b. Basic Language Structure

The language is free form in structure. Information is conveyed in the language through a series of entries which are punched on cards. An entry may consist of a single card such as would be used to specify a built-in network element. Certain entries require several cards and may include other entries as subentries. Such multiple card entries must always be terminated with a card containing the word END. The END card normally terminates only the most recently introduced unterminated entry. It is possible to terminate several entries simultaneously by using the word END followed by an appropriate key word. This will terminate all entries up to and including the most recently introduced unterminated entry which was initiated by that keyword.

The final card in a CSL deck is one of the following two cards:

```
END
END DECK
```

where the END DECK card may be used to terminate all previously unterminated entries.

Although the language structure is hierarchical due to the permissible nesting of entries, an indented structure is not required to delimit the extent of the various entries; this is accomplished by the END card. However, examples given in this report will be displayed in indented form to visually indicate the relationships between the various sections.

The order of entries is arbitrary except where a particular execution sequence must be specified.

Throughout this description of CSL there will be references made to global and local definitions, and to main network description notation. A global definition is a definition which is made on the outermost hierarchical level, i.e., it is made through an entry which is not contained inside of another entry. A definition which is contained within the context of another entry is a local definition; it applies only within the entry which contains it, and to any entries nested inside of that entry unless they, in turn, contain an even more local definition of the same quantity. Thus, the most local definition of a quantity always applies, and a definition never applies outside of the entry which contains that definition. In the absence of any local definition, the global definition applies; if the global definition is not present, a default definition may apply.

Since the network may be built up from nested subnetworks, there are usually several names which may be used to refer to a quantity which resides in the innermost nest, depending upon the nest level from which the reference is made. The main network description notation refers to the name which must be used to make the reference from the outermost level, i.e., that level which contains all nests.

c. Card Punching Conventions

Cards are punched in free format, beginning with any column. The language is free form in structure. The blank may be used as an optional character which is ignored except when it occurs in the text of a comment.

Punching on a card may begin in any arbitrary column for all entries. Continuation from one card to the next is accomplished by punching any of the characters +-,/(or = as the last non-blank character on the card to be continued (this rule does not apply to comment cards).

d. Comment Cards

Comment cards always start with the character $ and cannot be continued from one card to the next. A comment card may contain arbitrary text. As many comment cards as desired may be included in the deck and they may be arbitrarily inserted. Embedded blanks in comment cards will be preserved.

Comments may occur at any point in the CSL input deck except within the body of a procedure definition, table definition, or curve definition.

e. Text Definition

A group of cards containing arbitrary information may be assigned a symbolic name and then used freely at any point. The text definition is accomplished in the following manner:

```
TEXTn
.
.  (One or more cards which comprise the text which is being defined)
.
END TEXT
```

TEXTn is the symbolic name of the group of cards which follow, where n is an integer. The card group is terminated by and does not include the ENDTEXT card. The card group may include any entry, including any associated END card, except ENDDECK and other text definitions. A text definition must precede any reference to that definition in the card deck.

Text definitions are referenced by:

```
USE TEXTn
```

An example of text definition is given by:

```
TEXT3
   R3,4-7=1.5
   C6,3-7=.009
   USE TEXT7
   R4,2-5=7.8
   END TEXT
TEXT7
   C12,5-8=.003
   R45,2-5=6.5
   END TEXT
```

Consider the following excerpt using the above definitions:

```
R43,K-L=7
USE TEXT3
C76,F-G=12
```

This excerpt is equivalent to:

```
R43,K-L=7
R3,4-7=1.5
C6,3-7=.009
C12,5-8=.003
R45,2-5=6.5
R4,2-5=7.8
C76,F-G=12
```

A text definition may be referenced as many times as desired. Text references may be nested as shown in the above example.

A USETEXT card may occur at any point in the CSL input deck after the corresponding text has been defined except within the body of a procedure definition, table definition, or curve definition.

2. NETWORK DESCRIPTION

a. Node Names

Network node names are composed of any arbitrary sequence of the alphabetic characters and the digits. The name may be of any length, provided that continuation from one card to the next is not required.

The digit 0 is reserved as the node name for the datum node; the user may also specify additional nodes as the datum node by including a DATUM card. For example, to specify the node GND as the datum node, include the card:

DATUM,GND

b. Element Specification

There are two kinds of network elements: elements which are built into the simulation module and elements which are defined by the user in terms of the built-in elements and other user defined elements. The user defined elements are called subnetworks in this report.

The general format for element specification consists of three fields in the following order: the element name, the network nodes to which the element is connected, and the element parameter value(s) specification. The last two fields are optional. The format can be expressed as follows:

Elname,Nodes=Values

The Elname field specifies the element name, consisting of a prefix composed only of alphabetic characters followed by an alphameric suffix. The prefix specifies the kind of element. The suffix must always begin with a digit.

The Nodes field is separated from the Elname field by a comma and specifies the names of the network nodes to which the element is to be connected. It is permissible to use the character / as a prefix for a node name to denote optional properties associated with that node for certain elements. The - character is used as a delimiter between node names. If the element is a system element, the first nodes listed are the output nodes. The Values field begins with the = character and contains one or more parameter values associated with the element specification. Multiple values are separated by commas. Default values are available in the language processing module so that missing values may be automatically supplied using the default values.

Character strings are permitted as values for specific elements. Each character string must begin with an alphabetic character and contain only alphameric characters.

All built-in elements have a parameter symbol associated with each value. This permits an alternate method of describing such elements. The alternate format permits selected parameter values to be specified, the missing values assuming their default values. The alternate format is:

Elname,Nodes,(Valuelist)

where Valuelist is enclosed by parentheses and consists of one or more value entries, separated by commas. Each value entry consists of the parameter symbol, an equal sign, and the value of the parameter.

For example, a core winding element may require specification of both the winding resistance and the number of turns; let these values be associated with the parameter symbols R and N, respectively. Such an element can be described in the alternate format by:

CW45,4-6,(N=10,R=.001)

The standard element description format could also be used, except that the parameter values must be listed in the proper sequence (in this case R followed by N):

CW45,4-6=.001,10

This description of CSL does not explicity describe the network elements which may be included in the CSL language. The form of CSL is such that an unlimited number of network element types can be used with the language, with arbitrary prefixes, numbers of nodes, numbers of values, and arbitrary parameter names. Thus, the choice of network elements and their specification in CSL depends upon a particular implementation of CSL and not on the CSL language in general. However, voltage and current sources are briefly described in the following section because of their use of complex notation, a general feature of CSL.

(1) Voltage and Current Sources

A voltage and current source may be specified as either a real source or a complex source. A real source is used to drive the large signal DC and transient solutions; during the frequency domain solution it appears as a short circuit for a voltage source and an open circuit for a current source. A complex source is used to provide sinusoidal excitation for a linear frequency domain solution; during the large signal solution and during the two port small signal transfer function calculations it appears as a short circuit for a voltage source and an open circuit for a current source.

A real source is described using the normal element description format:

E56,G-M=3.8

A complex source requires two values to describe it. The two values are separated by a comma and the value pair must be enclosed by parentheses. One of the words CØMPLEX, RADIANS, or DEGREES may optionally precede the parenthesized pair. For example:

```
J78,B-U=CØMPLEX(4,8)
J78,B-U=RADIANS(5,1.2)
J78,B-U=DEGREES(5,65)
J78,B-U=(5,65)
```

If the word CØMPLEX appears, the first value is the real component, the second value is the imaginary component. If the word RADIANS appears, the first value is the magnitude, the second value is the phase angle in radians. If the word DEGREES appears, or if no word appears, the first value is the magnitude, the second value is the phase angle in degrees.

c. Subnetwork Definition

The user may define subnetworks which are composed of built-in elements and/or other subnetworks. Thus, it is possible to nest subnetworks. The format for subnetwork definition is identical to that for element specification except that SUBNET is used as a keyword ahead of the name field, the name field contains only the prefix, and dummy node and formal parameter names are used.

The subnetwork prefix must contain only letters and must not conflict with any other defined name which may start an entry. The formal parameter names are composed of alphameric characters and must begin with a letter. The formal parameter names must not conflict with any other defined name which can be referenced in a mathematical expression.

When formal parameter names are replaced during subnetwork references a character string replacement occurs. Thus, both values and character strings can be passed as formal parameters. Although only alphameric formal parameter names are permitted, the corresponding character string in the referencing expression is practically unlimited.

Subnetwork definitions may include element references, subnetwork references, subnetwork definitions, function definitions, table definitions, procedure definitions, datum node definitions, real variable definitions, and USETEXT entries. All definitions apply only to the subnetwork being defined and any subnetwork which is contained within that subnetwork, unless more local definitions are made within those descendant subnetworks. Each subnetwork definition is concluded with an END SUBNET card.

An example of a subnetwork definition is:

```
SUBNET,TR,J-K-L=A,BC
   LCB1,J-5=BC
   LCB2,5-3=BC
   C1,3-L=A
   R1,5-L=2
   R2,K-5=100
   F9(R)=R**2*EXP(-ABS(R/50))+35
   SUBNET,LCB,1-3=BC
      LCA1,1-2=BC
      LCA2,2-3=BC
      END SUBNET
   SUBNET,LCA,A-C=BC
      C1,B-0=TABLE3(BC(TIME))
      C2,C-0=5.67E3
      END SUBNET
   L1,A-B=F9(LCB1.LCA2.C1)
   L2,B-C=45.8
   TABLE3
     0,20
     5,25
     10,35
     15,36
   END SUBNET
```

This example illustrates the nesting concept. Subnets LCA1 and LCA2 are nested inside of subnet LCB, and subnets LCB1 and LCB2 are nested inside of subnet TR. A reference to any nested value, node, function, table, real variable, or procedure can be made from any higher nest level by constructing a compound name which traces the nesting process, using the dot as a separator between nest levels. Thus, in subnet TR the value field of inductor L1 references the capacitor C1 which is contained in subnet LCA2 which is contained in subnet LCB1; this is done by using LCB1.LCA2.C1 as the capacitor name.

If a datum node name 0 appears, it will be treated as being the global datum node.

A subnet may be used in the network simply by describing it as an ordinary network element. Using the subnet TR as an example:

```
TR2,1-3-0=TIME,SQRT
TR5,2-7-6=TIME**2,LØG
```

d. Model Definition

CSL permits the user to define models which may be stored in a model library. The model may only be defined globally and must not contain references to other models. A model definition is introduced by a card which contains the keyword MODEL, the model prefix name, the external dummy node names, and the model parameter names. This is followed by additional cards which define the model in terms of other CSL entries. The model definition may include element references, subnetwork definitions and references, function definitions, table definitions, procedure definitions, datum node definitions, and real variable definitions. The model concludes with an END MODEL card.

The model prefix must contain only letters and must not conflict with any other defined name which can start an entry. The model parameter names are composed of alphameric characters and must begin with a letter. The parameter names must not conflict with any other defined name which can be referenced in a mathematical expression.

An example of a model definition is:

```
MØDEL,DIØDE,1-2=RB,IS,TH,GC,W,C,VZ,N
   J1,3-2=IS*(EXP(TH*V.J1)-1)
   C1,3-2=C/(1-V(C1)/VZ)**N+TH*(J1+IS)/W)
   R1,1-3=RB
   R2,3-2=RC
   DEFAULT(RB=.001,IS=1E-8)
   END MØDEL
```

The model definition may include a set of default numerical values for the model parameters. This is done with the DEFAULT card as shown in the above statement. All values which are not assigned specific default values automatically assume a default value of zero. Alternatively, the DEFAULT values may be listed without the parameter names, using the same order as the parameter names are specified on the MODEL card:

```
DEFAULT=.001,1E-8,0,0,0,0,0,0
```

A model may be assigned a set of parameter values by means of a DEVICE entry. These parameter values are assigned a device name, permitting the parameter set to be referenced by the device name. All parameters which are not assigned specific values through the DEVICE entry will remain at the default values for the model with which the parameters are associated. The DEVICE entry occurs globally and lists the keyword DEVICE, the alphameric name to be assigned to the parameter set, the model prefix, and the parameter values. For example:

```
DEVICE,1N914,DIØDE=.001,1E-9,30,1E10,.7,3,1,.5
```

Here the parameter values are in the same sequence as the parameter names are listed on the MODEL card. If some other order is desired, or if only certain parameter values are to be specified, the alternate value format may be used in which the parameter names are specified:

```
DEVICE,1N914,DIØDE(TH=30,W=.7,RB=.001)
```

A model may be incorporated into the network description on any hierarchical level. This is done by considering it to be a standard element, using the model prefix followed by an alphameric suffix which begins with a digit. The value field specifies the name which has been assigned to the parameter set for the model; if no parameter set has been specified, the value field is not used. For example:

```
DIØDE1,56-A=1N914
SENSØR12,A-B
```

3. MATHEMATICAL NOTATION

a. Values

A value may be specified by either a number or a mathematical expression.

b. Numbers

Numbers may be written with optional magnitude sign, exponent sign, exponent, and decimal point. The exponent portion, if used, is introduced with the character E. Examples are:

```
25
+2.34
-35.78
4.E+6
49.789E-3
65E7
```

c. Mathematical Expressions

Mathematical expressions may be constructed using the rules of FORTRAN to represent values. A mathematical expression may be constructed using numbers, element parameter values, global variables, real variables, response variables, user functions, procedures, tables, and mathematical functions.

d. Element Parameter Values

All built-in element parameter values may be referenced. If the element has only a single parameter value, the value is referenced by simply using the element name. If the element has multiple parameter values, a particular parameter value may be referenced by constructing a compound name consisting of the element name, a dot, and the symbol for the parameter of interest (if only the element name is given, the value of the first parameter in the standard parameter order will be obtained). For example:

```
R3    gives the resistance value of resistor R3
CW6.R gives the resistance value of core winding CW6
CW6.N gives the number of turns for core winding CW6
CW6   gives the resistance value of core winding CW6 (by default)
```

e. Global Variables

Certain names are reserved for global variables, have global meaning, and may be used as desired. Examples are TIME and FREQ for the time and frequency variables.

f. Real Variables

The user may define real variables using names which do not conflict with reserved names in the language. The definition is accomplished by a mathematical equation where the left side is the real variable name being defined and the right side is a mathematical expression. All arithmetic is performed with real numbers. An example is:

```
FØRCE=M*A
```

Real variable names are composed of alphameric characters and must begin with a letter. The names must not conflict with any other defined name which may start an entry or be referenced in a mathematical expression, or the keywords OBJ, LINLOG, LOGLIN, LOGLOG, POLAR, NYQUIST, and FINAL.

g. User Functions

The user may define functions which include an argument list containing formal parameter names. The right hand side of the function definition is a mathematical expression of any complexity which is written in terms of the formal parameter names, other quantities, and other functions as desired. The function reference specifies the quantities which are to replace the formal parameter names during the computation of the function value.

Function names are composed of alphameric characters and must begin with a letter. The names must not conflict with any of the standard mathematical functions available in CSL, the keywords COMPLEX, RADIANS, or DEGREES, or any of the special functions and modifiers available in CSL, namely, SENS, PT, PV, IV, D, S, Q, E, F, N, A, B, Y, Z, I, V, U, and P.

The formal parameter names are composed of alphameric characters and must begin with a letter. The names must not conflict with any other defined name which can be referenced in a mathematical expression.

An example of a function definition is:

```
F56(A,B,C)=A*CØS(ANGLE)+B/TIME+C
```

This function may then be referenced as in this example:

```
C23,6-9=F56(3,N(3),F2(TABLE2(X13/I(C5))))
```

h. Procedures

The user may specify modules of FORTRAN code called procedures as part of the network description text. A procedure is introduced by a card containing the word PRØCEDURE and the procedure name, followed by one or more formal parameters. This is followed by the text of the procedure. The value of the procedure is specified by a replacement statement which has the procedure name on the left side. Control is returned to the calling expression by the RETURN card. The procedure concludes with an END card.

Procedure names are composed of alphameric characters and must begin with a letter. The names must not conflict with any of the standard mathematical functions available in CSL, the keywords COMPLEX, RADIANS, or DEGREES, or any of the special functions and modifiers available in CSL, namely, SENS, PT, PV, IV, D, S, Q, E, F, N, A, B, Y, Z, I, V, U, and P.

The formal parameter names are composed of alphameric characters and must begin with a letter. The names must not conflict with any other defined name which can be referenced in a mathematical expression.

An example of a procedure definition is:

```
PRØCEDURE, MAXF(A,B,C)
      IF(A.GT.B) GØ TØ 200
      MAXF=C(A-B)/N(1)
      RETURN
200   MAXF=C(A+B)*N(1)
      RETURN
      END
```

The procedure is written in terms of network quantities and other variables. Note that all values must be passed through the formal parameters unless the value is known from the level of network description on which the procedure appears. Character string replacement may be accomplished through the formal parameters.

A procedure may be referenced by simply including it as part of a mathematical expression, substituting the desired quantities and character strings for the formal parameters. For example:

```
G=MAXF(N(3),N(4),TABLE3)/MAXF(N(5),N(6),TABLE7)
```

During computation the first call to MAXF uses N(3) and N(4) as the values for A and B, and replaces the symbol C in the procedure definition with TABLE3; similarly the second call to MAXF uses N(5) and N(6) values and replaces C with TABLE7. The quantity N(1) must be defined on the nesting level at which the procedure appears.

i. Tables

The language has facilities for describing both one and two dimensional tables. Linear interpolation between listed values is used. Out of bounds values will be determined by extrapolating the slope of the table function at the appropriate boundary. Only numbers may be used to express the numerical values in the table definition.

The format for a one dimensional table is:

TABLEn

x_1, y_1

x_2, y_2

.
.
.
.

x_m, y_m

where: x_k = the kth coordinate numerical value corresponding to the table argument

y_k = the kth numerical value of the table corresponding to x_k

Periodic or repetitive tables may be specified by inserting the letter R for the value of y_m. This indicates that the table is to be repeated indefinitely with a period $x_m - x_1$.

A two-dimensional table describes an arbitrary function of two arguments. The format for two-dimensional table definition is:

TABLEn, 2D

y_1, y_2, ... y_m

x_1, z_{11}, z_{12}, ... z_{1m}

x_2, z_{21}, z_{22}, ... z_{2m}

.
.
.

x_k, z_{k1}, z_{k2}, ... z_{km}

where: $y_1, y_2, \ldots y_m$ are the numerical values of the second argument used to define the table values z

$x_1, x_2, \ldots x_k$ are the numerical values of the first argument used to define the table values z

z_{ij} is the numerical table value corresponding to the specified values of x_i and y_j

Examples of one and two dimensional table references are:

```
TABLE56(TIME)
TABLE35F(CX,CZ)
```

j. Time Domain Response Variables

The following forms are used for the time domain response variables where Elname is the element name and Node is the node name involved.

```
N(Node)     for nodal voltage response
I(Elname)   for element current
V(Elname)   for element voltage
P(Elname)   for element power
E(Elname)   for element stored energy
F(Elname)   for magnetic flux in element
Q(Elname)   for electric charge in element
```

k. Frequency Domain Response Variables

In the frequency domain the network solution is generally complex; thus one must be able to differentiate between the real and imaginary components of the response. This is done by appending a character denoted by x after the response designation character. The letter x may represent the following characters:

```
x = M     for magnitude
    D     for phase angle in degrees
    P     for phase angle in radians
    R     for real component
    I     for imaginary component
```

The response variables for the frequency domain are:

```
Nx(Node)   for nodal voltage response
Ix(Elname) for element current
Vx(Elname) for element voltage
Px(Elname) for element power
```

The network must include at least one complex source in order to produce a nonzero network response for the above response variables.

The two port small signal transfer functions are also available in the language. These assume either open circuit current or short circuit voltage excitation with a unit amplitude sinusoid of zero phase on the input port, and specify either the open circuit voltage response or the short circuit current flow across the output port. Thus, one can determine voltage gain, current gain, transfer and driving point admittance, and transfer and driving point impedance by a suitable set of port terminal conditions. The transfer functions are denoted by:

```
Ax(a-b/c-d)  for voltage gain
Bx(a-b/c-d)  for current gain
Yx(a-b/c-d)  for transfer admittance
Zx(a-b/c-d)  for transfer impedance
```

where the output port is designated by node a with respect to node b, the input port is designated by node c with respect to node d, and x has the same meaning as before.

Abbreviated versions of this notation are available. If either node b or node d is the datum node it may be left out. If nodes c and d are identical to node a and b, respectively, the c and d nodes may be left out. Equivalent examples are:

```
ZM(2-0/2-0)
ZM(2-0/2)
ZM(2/2)
ZM(2)
ZM(2-0)
```

1. Network Sensitivity Matrix

The network sensitivity matrix may be calculated in the large signal DC and transient solution and in the frequency domain solution. The network sensitivity matrix consists of the partial derivatives of a vector of network response variables with respect to a vector of network parameters. An example of the notation for the network sensitivity matrix is:

SENS(N(5), I(L5), DISPLACEMENT/FØRCE, R66)

where N(5), I(L5), and DISPLACEMENT are the response variables and FØRCE and R66 are network parameters. This example produces a 3x2 matrix of sensitivities.

m. Value Modifiers

Any element parameter, global variable (except FREQ), real variable, or time domain response variable may have a modifier prefixed to it to designate related time or iterative values. Letting Name denote the symbolic name of the quantity for which a related value is desired, we have the following forms available:

Name	for current value
PT(Name)	for value at previous time step
PV(Name)	for value at previous iteration
IV(Name)	for initial value (at time = 0)
D(Name)	for time derivative value
S(Name)	for time integral value

4. SOLUTION CONTROL

The network description language which has been presented only describes the network configuration of interest. It does not provide for any calculation of network response. Response calculation is initiated by specifying one or more solution entries. The solution entries are executed in the order in which they are encountered in the input deck. The solutions available are nominal response, Monte Carlo, optimization, and worst case.

A solution entry may require additional information to enable proper execution. This information may be specified by run controls, initial conditions, network parameter value changes, swept and parametric variables, the BØUNDS entry, and the CURVE entry. These topics are discussed before the solution entries are presented.

In the discussion which follows the term normal value is used. The normal value is the value of a quantity which is used when no change in value is specified by the user in a particular solution.

a. Run Controls

CSL permits the user to specify run controls globally, within nominal solutions, Monte Carlo solutions, optimization solutions, and worst case solutions, and within state entries of optimization solutions. Specific run controls are not included in this discussion since the CSL language has the capability of handling an unlimited set of run controls. Thus, the choice of a particular set of run controls depends upon a particular implementation of CSL and not the CSL language in general.

The grammar of the CSL language permits six basic forms of run controls. The basic forms are:

```
Keyword
Keyword = Keyword
Keyword = Integer
Keyword = Real number
Keyword = Math expression
Keyword(Boolean expression)
```

The Boolean expression may be of any complexity using FORTRAN rules for combining Boolean operators, arithmetic relational operators, and mathematical expressions.

b. Initial Conditions Specification

The user may specify the initial conditions to be assigned to any network variable for which initial conditions has meaning. This is done by means of the INITIAL CØNDITIØNS entry which may be specified either globally or locally within a solution entry.

The first card of the entry contains the words INITIAL CONDITIONS; the last card is an END card. The intervening cards specify the network variable name (using main network description notation) and the value of the initial condition. For example:

```
INITIAL CØNDITIØNS
   V(C3) = .075
   I(L3) = 1200.5
   VEL = 2.9
   END
```

c. Network Parameter Value Changes

Network parameter values may be changed to a different value during a particular solution by specifying the parameter name and the new value. The parameter name corresponding to the main network description notation is used. The new value may be given as either an actual value or as a relative value. A relative value is denoted by a number suffixed with an asterisk; the actual value is then the product of the normal value and the specified number.

Examples are:

```
R12 = 350/C3
C23 = .85*
D2.TH = 1.89*I(D2)
GT6.FY7.T35.BN = .55*
FREQ = 367
```

d. Swept and Parametric Variables

Quantities may be stepped through a series of values during a solution by designating them as swept and parametric variables. Only one quantity may serve as a swept variable. There may be as many parametric variables as desired. The parametric variables are held at a particular value while the swept variable is swept through its range. Then the parametric variables are stepped to their next value and the swept variable is again swept through its range. In this way, a family of curves may be generated through the use of these two types of variables.

An example of a swept variable and two parametric variables is given by:

```
FREQ = .005, (25LØG), 1.5*
*R12 = 1, (3), 4, 10, (2LØG), 20, 35
*R34 = 10, (2), 15, 20, 50, 100, (2), 200
```

Parametric variables are distinguished from swept variables by prefixing them with an asterisk. Thus, in the above example, the swept variable is FREQ and the parametric variables are R12 and R34.

Only numbers may be used in specifying the values for swept and parametric variables. Note that some numbers are enclosed by parentheses while others are not. The numbers which are not enclosed by parentheses represent specific values to be assumed by the variable. If these numbers are suffixed with an asterisk, the relative value is indicated; the actual value is then the product of the normal value and the relative value.

The number in parentheses indicates a step number, which is the number of steps involved when incrementing from the preceding value to the succeeding value. Normally, linear steps are used; however, if the word LØG is appended to the number in parentheses, logarithmic steps are used. The written number sequence may not begin or end with step numbers, and two step numbers may not occur in succession in the sequence.

When more than one parametric variable is used, the total number of values which will be assumed for each must be the same.

Thus, in the example shown above, R12 assumes the values of 1, 2, 3, 4, 10, 14.14, 20, and 35; while R34 assumes the values of 10, 12.5, 15, 20, 50, 100, 150, and 200.

If TIME is specified as either a swept or parametric variable, the user must insure that the sequence of time values which he specifies is monotonically increasing. If the termination run control has been specified, the word FINAL may be used to specify the value of TIME when the termination condition is first satisfied:

```
TIME = 0, (3), 600, FINAL
```

e. BØUNDS Entry

The user may specify the upper and lower limits which network parameters may assume by means of the BØUNDS entry. The BØUNDS entry may be specified either globally or locally within a solution entry. The BØUNDS entry is required by the Monte Carlo, optimization, and worst case solutions.

The first card of the BØUNDS entry is the word BØUNDS; the last card is an END card. The intervening cards specify network parameter names (using main network description notation) and their minimum and maximum value limits. Relative value designations may be used; in such a case the actual value is the product of the normal value and the relative value.

If the BØUNDS entry is to be used in conjunction with a Monte Carlo solution, one may specify distribution shapes for individual parameters. The available distribution shapes are denoted by the words GAUSS, RECT, TRI, and CURVEn.

The GAUSS shape is a Gaussian curve with the limit values specified at the 3σ points.

The RECT shape is a uniform distribution between the two limit values.

The TRI shape is a triangular distribution with the limit values corresponding to the bottom corners of the triangle, and the normal value corresponding to the apex of the triangle. Obviously, the normal value must fall between the two limit values.

The CURVEn shape specifies an arbitrary shape corresponding to the named CURVE entry. The first limit value corresponds to the minimum value, the second to the maximum value in the appropriate CURVE entry.

If the distribution shape is not specified the RECT shape will be used by default. The distribution shape information is ignored for optimization and worst case solutions.

Normally, the distribution shape is applied in linear space. The user may elect to apply the distribution shape in logarithmic space rather than linear space; this is done by including the word LØG following the distribution shape specification.

An example of the BØUNDS entry is:

```
BØUNDS
   R2=12,15
   GT7.R1=0.5*,1.5,CURVE3,LØG
   R6=PRØCEDURE XYZ(12,J),.4*,RECT
   END
```

f. CURVE Entry

The CURVE entry specifies a weighted curve which may be used by the optimization and Monte Carlo solutions. The format is:

CURVEn

x_1, y_1, w_1

x_2, y_2, w_2

.
.
.

x_m, y_m, w_m

where: x_1, x_2, etc. = values of the independent variable.

y_1, y_2, etc. = values of the curve corresponding to x_1, x_2, etc.

w_1, w_2, etc. = values of the curve weights corresponding to x_1, x_2, etc.

The specification of the weights for any point is optional. If the weight w_i is not specified, it is assumed $w_i = 1$.

When the curve is utilized by the Monte Carlo solution, the curve weights are ignored.

g. Nominal Solution

The nominal solution consists of one or more network response calculations corresponding to a prescribed set of values of network parameters. The normal network parameter values are used in the calculation except where specifically superseded in a given nominal solution.

Each nominal solution is specified by a RUN entry which must include an output specification. The first card of the entry is:

RUN

This card is followed by one or more cards which specify output, changes in the values of network parameters, swept and parametric variables, initial conditions, and run controls. The entry is terminated by an END or ENDRUN card.

The main network description notation is used in referring to the network parameters.

(1) Output Statements

The output statements specify which response variables are to be calculated, and how the results are to be presented. Since it is possible to vary network parameters over a range of values, the output statements can include plotting specifications.

Quantities which may appear as output include all network parameters, global variables, real variables, response variables, and mathematical expressions involving these quantities.

(a) Print Statement

This statement specifies the quantities which are to be printed. An example is:

PRINT, R1, HT5.T35.BN, N(1), AD(1-0/2-0), N(2), FREQ

This statement causes the printing of the quantities which are named in the statement.

Normally the printed listing uses the symbolic names of the requested quantities for labeling purposes. The user may specify a different label for a quantity by including that label, enclosed by parentheses, immediately following the symbolic name. For example:

PRINT, N(3)(ØUTPUT), R6(GAIN), L3

(b) Plot Statement

This statement specifies the quantities to be plotted as well as the coordinate system to be used in plotting. The independent variable is specified last, and is introduced by a / character. If no coordinate system is specified, a linear system is used. Each plot statement produces a separate graph. Examples of plot statements are:

PLØT, N(L)
PLØT, LINLØG, AM(1-0/2-0), ZM(1-0/1-0), /FREQ

As many dependent variables as desired may be listed. Obviously, only one independent variable may be named.

Coordinate systems are specified after the word PLØT. Choices are:

No specification	Linear in both dependent and independent variables.
LINLØG	Linear dependent and logarithmic independent variables.
LØGLIN	Logarithmic dependent and linear independent variables.
LØGLØG	Logarithmic in both dependent and independent variables.
PØLAR or NYQUIST	Dependent variable is plotted on polar coordinates (linear magnitude and phase) as a sequence of points corresponding to the swept variable values.

For PØLAR or NYQUIST plots an independent variable is never specified because both coordinates refer to the dependent variable. The format is shown by this example:

PLØT, PØLAR, AM(3-0/2-4), ZM(2-0/3-5)

Obviously, the usage of PØLAR coordinates is restricted to the frequency domain response variables. The magnitude of the frequency domain response variable is always specified when using PØLAR coordinates.

If no independent variable is specified for a given graph, the swept variable will be used automatically. For example:

```
RUN
   R3=1, (10), 20
   PLØT, I(L3)
```

will produce a plot of I(L3) versus R3.

Normally the plotted output uses the symbolic names of the requested quantities for labeling purposes. The user may specify a different label for a quantity by including that label, enclosed by parentheses, immediately following the symbolic name. For example:

```
RUN
   PLØT, N(3)(ØUTPUT), R6, /I(L3)(CURRENT)
```

h. Monte Carlo Solution

The Monte Carlo solution synthetically constructs a large number of networks using parameter values chosen from specified statistical distributions, analyzes the performance of each of these networks, and then summarizes the results as a set of performance statistics.

The network parameters which are to be statistically varied, the statistical distribution shapes, and the minimum and maximum parameter values to be associated with these shapes are specified in the BØUNDS entry which may be specified globally or within the Monte Carlo entry.

The first card of the entry is:

```
RUN MØNTE CARLØ
```

This is followed by output specification cards and optional BØUNDS entry, run controls, initial conditions, and parameter value changes. The entry concludes with an END or ENDRUN card.

Output quantities for the Monte Carlo Solution may include all network parameters, global variables, real variables, response variables, and mathematical expressions involving these quantities.

Output is available in both printed listing and plotted form. If printing is specified, the tabulation includes each individual network calculation. Plotted information is presented in histogram form.

An example of a Monte Carlo entry is:

```
RUN MØNTECARLØ
     R7=1.2*
     L5=L3/2
     PRINT, R1, N(1)
     PLØT, T1.RBB
     PLØT, AR(1-0/3-0)
     END
```

i. Optimization Solution

The optimization solution attempts to attain a desired optimum by a minimization process which varies specific network parameter values within an allowable parameter space. The minimum and maximum limits for the network parameter values which are varied to produce the optimum condition are specified by the BØUNDS entry. The values of the network parameters which produce the optimum condition automatically replace the normal values of these parameters for subsequent solutions.

The optimization solution is controlled by the optimization entry. The first card of this entry is:

```
RUN ØPTIMIZATIØN
```

This is followed by cards which specify operational state information and optional BØUNDS entry, run controls, initial conditions, and parameter value changes. The entry concludes with an END or ENDRUN card. The BØUNDS entry must not be introduced as a subentry within an operational state. Run controls and parameter value changes may be introduced as subentries within the optimization entry or within an operational state entry.

(1) Operational States

The operational state information includes changes in network parameter values, objective function(s) to be minimized, run controls, initial conditions, and the swept variable (if any). Parametric variables

are not permitted. There may be many different operational states specified, each corresponding to a different mode of operation for the network.

Each operational state is introduced by the card:

STATE

This card is followed by one or more cards which give the details for that particular operational state. The operational state is terminated by an END or ENDSTATE card.

In the actual optimization process all of the operational states are considered "simultaneously." Thus the optimized network gives the "best" performance in each of the states specified.

The usage of the operational state is best explained by referring to an extended example:

```
RUN ØPTIMIZATIØN
    C3=69
    STATE
        V6=-7
        S12=0
        TIME=0
        ØBJ=N(1)/I(L5)
        END
    STATE
        ØBJ=I(L5)**2, CURVE3, .25
        S12=1
        ØBJ=N(1), CURVE5, .50
    STATE
        ØBJ = AR(1-0/3-4), CURVE2, .35, X
        TIME=47
        FREQ=10, (20LØG), 100
    END RUN
```

Two different forms of objective function statements are permitted. One form, called the point form, is used when only a single value for each network response variable is calculated; the other form, called the curve form, is used in conjunction with a swept variable.

The point form is shown in the first operational state in the example above. It consists of the word ØBJ, an equal sign, and a mathematical expression whose value is to be minimized. This mathematical expression is evaluated only once during a particular response calculation. If the DC steady state response is specified, then it is evaluated at the DC operating point. If the transient response is specified, it is evaluated at the termination of the transient calculation. There is no restriction on the form of the mathematical expression.

The curve form is shown in the second and third operational states in the above example. It requires a series of values for some mathematical expression for various values of a swept variable. If the swept variable is TIME, a set of time values must be supplied.

The curve form is specified, in order, by the word ØBJ, an equal sign, a mathematical expression, the name of a curve, a number specifying a weighting constant, and an optional fitting parameter.

The curve form actually specifies a curve fitting process. The curve name refers to the definition of a specific curve shape which is included somewhere else in the input. It is required to fit the values of the mathematical expression as closely as possible to the curve values in the least squares sense. The independent variable for the mathematical expression values and the curve is always the swept variable.

The curve is normally required to be fit as exactly as possible in the least squares sense. However, by including the operational character X at the end of the curve form statement, it is possible to do the fitting to within a linear transformation. This transformation will be chosen by the program and generally its value is of no interest to the user.

There is no restriction on the number of objective functions which may appear under a given operational state. Both point form and curve form objective functions may not be used in the same operational state.

The optimization solution minimizes the sum of the values of all objective functions for all specified operational states.

j. Worst Case Solution

The worst case solution is specified by the worst case entry. The first card of the entry is:

RUN WØRST CASE

This is followed by cards which specify network parameter value changes, initial conditions, and output requests. The END card terminates the entry. The transient response calculation cannot be calculated in the Worst Case Solution.

The BØUNDS entry must be included either globally or locally. The output quantities for which the worst case solution is desired are listed in a PRINT statement. The worst case response in both the high and low directions is calculated for all output quantities using the appropriate limit values for the parameters specified in the BØUNDS entry. Plotting of worst case output is not available.

SECTION IV

INITIAL CSL TRANSLATOR DESIGN

1. PHILOSOPHY OF THE CSL TRANSLATOR

The CSL Translator performs the automatic translation of source text in the CSL language to its equivalent in the NET-2, SCEPTRE, and CIRCUS-2 target languages. It is desirable that the translator be more than just a hard coded translator between these languages. For instance, the form of CSL may change as new features are added or existing features are modified. Translation between other languages may be desired at a later date. These considerations dictate the primary design philosophy of the CSL translator; it will be modular and table driven to the greatest extent possible. Furthermore, the tables which control the translator will be capable of formal description, using a higher level language specifically designed for such description. In this way, as much of the essential description of the translator as possible can be described using formal techniques, permitting flexibility in the modification of the translator required by lexical, grammatical, or semantic changes in any of the languages handled by the translator.

It is well known that a language can be described by three kinds of components: lexical, grammatical, and semantic. Lexical components comprise the "words" of the language, the vocabulary which is available for constructing sentences or text streams in the language. These lexical components may be combined through a set of rules, known as the grammar of the language (also called the language syntax), to produce properly constructed sentences in the language. Finally, the language semantics give meaning to the sentence which has been constructed so that specific information is conveyed by the sentence.

The language translation problem is concerned with all three aspects of the language. Although the semantic aspect is of primary concern during actual translation, it is also necessary to include both the lexical and grammatical aspects since they provide a means for representing the source and target texts. The formal recognition of these three aspects of a language was given considerable attention during the CSL Translator development effort. They have been treated in modular fashion, both in the actual software design and in this report, permitting changes in one component without adversely affecting the others.

A number of formidable problems have appeared during the translator design and development, preventing the realization to date of a completed translator. Many of these problems were successfully solved, while others still remain. This report documents various aspects of the project. The remainder of this section discusses initial efforts at producing a translator. These efforts were subsequently abandoned and a more comprehensive approach was pursued, as described in subsequent sections of this report.

2. INITIAL CSL TRANSLATOR

During the early stages of the translator project, software was developed which has the capability of translating a subset of CSL to a subset of NET-2. As attempts were made to extend the concepts which had been developed to larger subsets of these languages, and ultimately to SCEPTRE and CIRCUS-2, the deficiencies of the approach which had been selected became sufficiently important that it was not feasible to continue translator development along those lines.

The CSL translator operates from a Backus-Naur Form (BNF) of both the source and target languages. The BNF description of a language specifies the grammar of the language. If the BNF description is carried to the character level, then the lexical description of the language may also be specified by the BNF. This approach was used in the initial translator design.

Specifically, a method of expressing the grammar and lexical aspects of the languages was developed, using a modified BNF notation which was amenable to being keypunched (this notation was somewhat more difficult to read than the notation which was later adopted and used in this report). A special program was written which transformed the BNF notation into a set of tables which was then used to direct the parsing of the text stream of the language of interest.

As the input stream was parsed, information was assembled into a canonic data structure (CDS). The CDS was stored as an n-ary tree structure in which sibling nodes represented alternative terms in the BNF productions, and descendant nodes represented nonterminal expansions in the BNF. The actual character strings which formed the input text appeared at the leaf level of the tree.

Each nonterminal in the BNF was assigned a numeric code which was stored in the tree structure. Thus, by traversing the tree, one could readily reconstruct the input text, as well as inspect the parse tree which resulted from parsing the text.

The CDS which resulted was generally too detailed and required more memory resources than needed for the essential information stored in it. Thus, the concept of tree pruning was devised. A special marker was embedded in the BNF as a prefix for certain terminal and nonterminal terms. The marked terms were designated as essential nodes in the CDS. After a subtree of the CDS had been formed, it was pruned by eliminating any serial nodes which did not correspond to marked terms, providing that the removal of such nodes did not destroy the basic structure of the CDS. Thus, the CDS was compacted by removing nonessential nodes, yet the hierarchical relationships, as expressed by the underlying BNF, were preserved.

The first phase of the translation process was now complete, that is, the source text stream had been parsed and a canonic data structure generated.

The next step involved transformations on the CDS in accordance with the specific translations to be made. This involved operations on the CDS tree which added, deleted, moved, and modified nodes, subtrees, and character strings at the leaves. The details depend specifically upon the languages involved. At the completion of this step a new form of the CDS existed, corresponding to the target language. In fact, the resultant CDS was exactly that CDS which would have resulted at the completion of the first phase if the target language had instead been the source language.

The target text was now generated from the target CDS by simply performing operations which were the inverse of parsing the target language. Again, a BNF table for the target language grammar was used which included term markers indicating essential nodes in the target CDS. The CDS tree was traversed, and as a result of traversal, a correspondence was established between the target BNF and the essential nodes of the target CDS. This resulted in a specific traversal of the BNF which in turn generated the target text as output.

A translator which was constructed using the above techniques which had the capability of translating the Monte Carlo entries of CSL into the corresponding entries in NET-2. Later, the translation was extended to include the CSL Run entry and Optimization entry, with translation into the NET-2 equivalents.

As this approach to the translator design continued several problems became apparent. First of all, the parsing operation was of the top-down type in which the parse tree is constructed by starting with the grammar goal symbol and attempting to build the tree down to the lexical symbol level (in this case, the character level). Thus the parser exhibited all of the problems of top-down parsers. Error detection was delayed until all possible parse tree constructions had been investigated, the parser was slow in operation (particularly in this case since parsing was carried down to the character level), and error recovery was extremely difficult. Certain BNF representations were found to trap the parser into dead end situations from which it could only report syntax error, when, in fact, there was no syntax error.

Problems were also evident in resolving keywords which could have several different semantic interpretations as well as represent identifiers, depending upon the context in which they are used. The magnitude of this problem can be seen by the complexity of the lexical analyzer which has been successfully developed for the CSL language, as shown in Appendix F. The initial approach to CSL parsing could not adequately handle the lexical analysis requirements demanded by the CSL language.

The CDS depended upon the particular BNF specification of the grammar for its topological structure. Thus, two equivalent BNF representations of the same grammar could generate substantially different canonic data structures from a topological point of view (the same information was available in both, but linked differently). Since the transformation routines required knowledge of the precise topological structure of the CDS, it can be seen that the BNF and the coding of the transformation routines were completely interdependent, precluding the goal of modular independence between these two parts of the translator. The problem was further compounded by the fact that this effect occurred on both the source and target sides of the translator.

Finally, the CDS transformation routines contained considerable overhead associated with the initial and final CDS formats; this overhead had no relation to the actual semantic translation which was to be accomplished. The transformation software tended to become very unwieldy because of this.

The extension of the initial approach to include larger subsets of the languages under consideration became unfeasible. A number of crucial problem areas had been illuminated and it was necessary to find solutions for them before continuing. Accordingly, this approach was abandoned and a new approach was developed. The new approach solved all of the problems encountered under the original approach and had far greater generality.

The new approach uses a flexible bottom-up parser coupled with a lexical analyzer to solve the problems associated with parsing, error detection, error recovery, and lexical analysis. The ability to couple semantic routines arbitrarily with the parsing operation permits the source canonic data structure to be independent of the BNF representation of the source grammar. The transformation from source to target canonic representation is accomplished by a generalized translation model through which data structures are manipulated through a BNF type description with associated semantic processing. The final translation from target canonic data structure to output strings in the target language is accomplished by syntax directed translations which can generate output strings in parallel with the parsing of the target canonic data structure.

When the new concepts had crystallized and their feasbility had been established, work on the original approach was suspended. Clearly the first requirement was the development of an LR(k) parser and a lexical analyzer to work with the parser. The manual construction of an LR(k) parser for a grammar of the size of the CSL grammar is clearly impractical. Therefore, effort was expended in producing an LR(k) Parser Generator which accepted the BNF description of the CSL grammar (or any other grammar for that matter) and generated the tables required by the parser for that grammar. Similar problems associated with the manual generation of the lexical analyzer tables resulted in the development of a Lexical Analyzer Generator which produced the lexical tables for the CSL language (or for any other language of interest).

For the CSL translator itself an LR(k) parser and a lexical analyzer are needed along with the tables which program them for a particular grammar and language. However, the LR(k) Parser Generator and the Lexical Analyzer Generator also required a parser and lexical analyzer; thus the entire system was developed in a bootstrap fashion, using a primitive parser and lexical analyzer with manually generated tables as a starting point, and building up to the final general purpose system.

Following the construction of the basic Parser Generator, its capability was extended to permit semantic routine linkages, error recovery, and syntax directed translations to be handled. The resulting programs provide for a language implementation system which has applicability far beyond the CSL translator.

SECTION V

A GENERAL TRANSLATION MODEL

This section and all subsequent sections of this report describe work accomplished in the realization of the CSL Translator following abandonment of the initial translator design discussed in Section IV. The general approach to translation is discussed in this section.

1. THE TRANSLATION MODEL

Translation may be viewed as a process of transforming a set of input data into a set of output data, according to some specified rules. In this report we are concerned with the translation of one artificial language into another; i.e., the translation of a source language into its equivalent target language representation. The source language of interest is the Common Simulation Language (CSL); the target languages are the problem description languages for the NET-2, SCEPTRE, and CIRCUS-2 programs.

Any valid expression in a language is called a sentence of the language. For purposes of this report a sentence in any of the languages under consideration is the complete problem description in that language; i.e., the entire CSL input deck or the entire output deck of the translator is considered as a sentence in the respective language. The complete set of all valid sentences in the language is the language itself.

Sentences are composed of a sequence (or string) of symbols. In English the symbols are the words and punctuation which make up a sentence. In the artificial languages under consideration the symbols represent node names, element names, keywords, delimiters, etc. The set of all symbols which may be used in composing a sentence in a language is called the vocabulary of the language. The symbols are called the lexical components of the language.

Sentences are composed by combining symbols into certain sequences according to a set of specific rules. These rules are called the grammar of the language. The syntax of a language refers to the grammatical structure of sentences in the language.

Thus, we see that the grammar of a language is the syntactic description of that language. Language is not useful however until the meaning of the sentence in the language can be described. The meaning of a sentence is given by the semantics of the language. The grammar contains no semantic information in itself. Syntactic analysis of a sentence permits one to determine if that sentence is a valid sentence in a particular language; it does not, in general, permit the translation of the sentence into an equivalent sentence in another language. Thus, it is seen that syntactic analysis permits recognition of a sentence in a given language, whereas semantic analysis is required to translate the sentence into another language.

The process of decomposing a sentence into its component parts (down to the symbol level) is called parsing. This operation is accomplished by a parsing algorithm (or simply, a parser) which uses a formal description of the grammar of the language as a program to direct the parse.

The parser operates on terminal symbols as input data. These symbols, however, are not the lowest level of information in which the sentence is written. Symbols are composed of characters just as words in English are composed of letters. Thus, it is necessary to have a means of generating symbols. This is accomplished by a lexical analyzer which accepts input at the character level and outputs symbols according to a prescribed set of rules. The rules for generating symbols generally depend upon the grammatical context in which they appear. Thus, the parser must have the ability to inform the lexical analyzer as to which set of rules are to be used in generating the next symbol required by the parser. The lexical analyzer then uses these rules to generate the next symbol.

The lexical analyzer is composed of one or more scanners, where each unique set of rules for forming symbols corresponds to a particular scanner. Thus, the parser designates the scanner to be used by the lexical analyzer, and the selected scanner has a particular set of rules associated with it which are used in forming symbols from the characters which are input to the scanner.

The scanners in the lexical analyzer receive characters as input information. These characters are supplied from a device called a character generator. The character generator is responsible for traversing a data structure and extracting characters to pass on to the lexical analyzer. Traditionally, the character generator extracts its characters from punched card images. However, in the model for translation which is being presented the characters are not restricted to keypunch characters, nor is the data structure restricted to card images. In fact, the character generator may be called upon to traverse any kind of data structure and extract characters which are totally unrelated to the keypunch characters. This concept is central to the generalized translation model which is used in the CSL translator.

We have noted that the translation process requires both syntactic and semantic analysis. The semantic analysis is coupled with the parsing operation and invoked at specified points during the parse by the parser. The semantic analysis is accomplished through a set of semantic routines. The semantic routines have access to information developed during the parsing process and are able to construct data structures which may be used as input to the character generator in a later phase of the translation process.

This leads us to the concept of translation phases. The translation can be considered to operate in a finite set of phases. In any phase a data structure exists which contains a sentence in a particular language; this language may be either the original CSL language or an intermediate language. Associated with this language is a set of grammatical rules stored in a parsing table. Thus, the execution of a particular phase consists of specifying a particular input data structure and a parsing table. The parser uses the parsing table to direct its syntactic analysis of the sentence contained in the data structure. In so doing, the parser needs symbols which it obtains through the designation of a particular scanner in the lexical analyzer. The scanner needs characters in order to form the symbols; these characters are obtained from a particular character generator associated

with the scanner. The character generator traverses the designated data structure and so obtains the characters which make up the sentence in the data structure. Finally, as the parse progresses, the parser invokes specific semantic routines. These semantic routines then construct a sentence in a new language in an output data structure. This completes the translation phase.

The final phase results in the construction of a sentence in the target language using an output medium to contain the sentence. This completes the translation process. The above technique can be extended to translations of any complexity. It has the virtue of being highly modular. The process can be formally defined at every step of the overall translation. Thus, documentation of the translation technique is easily achieved.

The parser and the scanners are driven by tables. Thus, the parser and the lexical analyzer with its associated scanners can be standardized software packages, independent of the application for which they are used; in fact, the lexical analyzer contains only a single scanner --- the multiple scanners are implemented simply by switching tables on the single scanner that actually exists. Similarly, the multiple parsers are in reality only one parser, with table switching employed to accomplish the multiple parser concept.

On the other hand, there actually are multiple semantic routines and character generators since these perform unique operations which are dependent upon specific data structures and semantics. It has not been found practical to implement a standardized semantic routine and character generator using tables to program their specific behavior.

A schematic representation of the translation model discussed above is shown in Figure 3.

2. GRAMMAR SPECIFICATION

From the foregoing brief description of the translation model it can be seen that all activity during a translation phase is dependent upon the action of the parser. Thus it is necessary to have a formalism for specifying

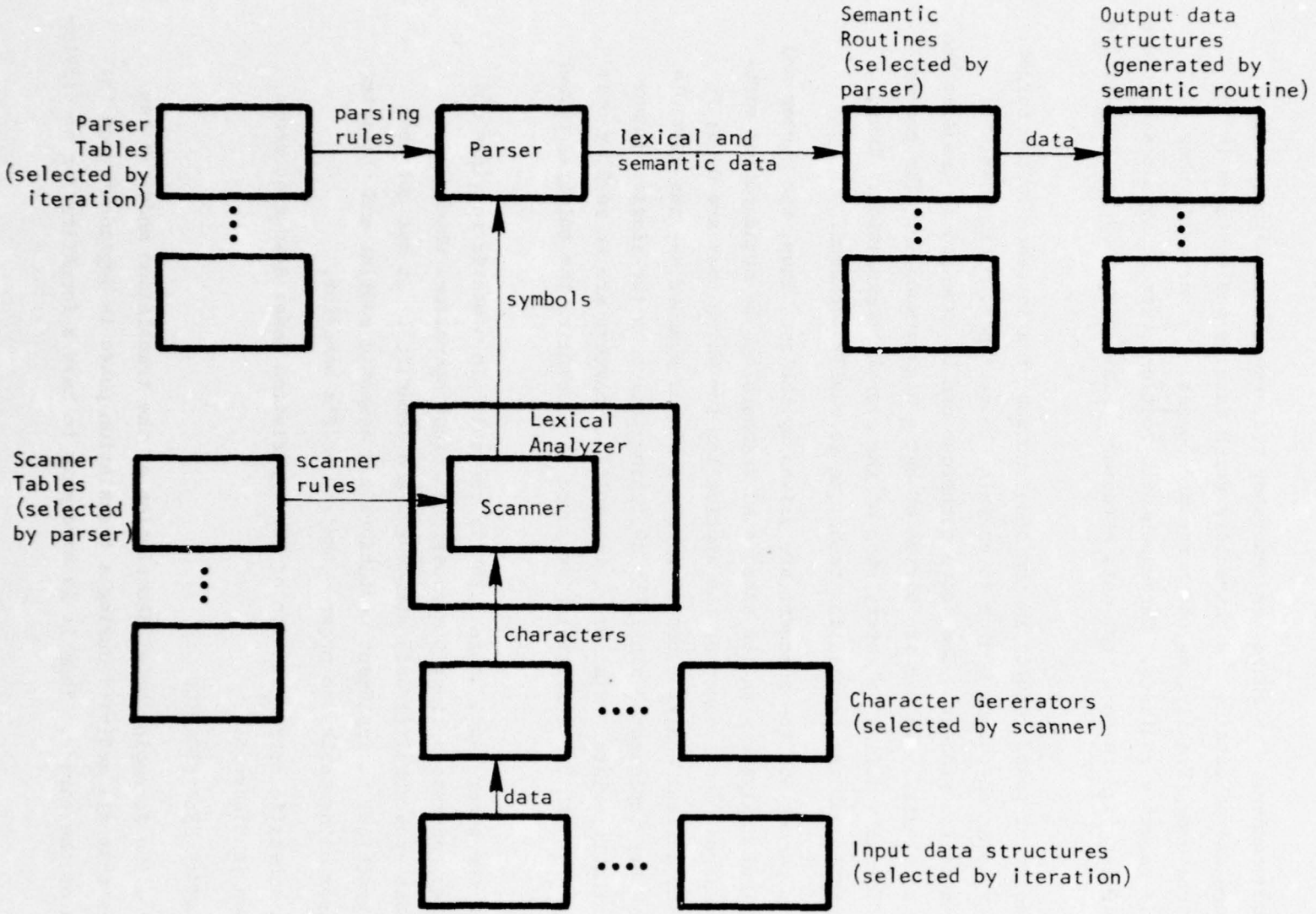


Figure 3. General Translation Model

the grammar of the language whose sentences are to be parsed. Through this formalism one can then construct the parser tables and thus control the actions of the parser.

The grammar G of a language can be expressed as a 4-tuple G = (N,T,P,S) where:

N = a set of nonterminal metalinguistic symbols

T = a set of terminal symbols

P = a set of productions or rules defining the nonterminals N

S = the grammar goal symbol

The set N is disjoint from the set T; i.e., a symbol cannot be used both as a terminal and nonterminal. Each nonterminal symbol is defined by one or more productions. A production is an ordered set (U, x) where U is a single nonterminal symbol called the production goal symbol and x is an arbitrary string of nonterminal and terminal symbols which define U. A particular nonterminal symbol S is known as the grammar goal symbol; all other nonterminal symbols are derivable from the grammar goal symbol.

A notation known as Backus Naur Form (BNF), also called Backus Normal Form, is particularly convenient for describing productions of the grammar. In BNF the ordered set (U, x) is written as

U = x

where the definitions of U and x are as above. Any language whose grammar can be expressed by productions of this form belongs to the class of context free languages, and its grammar is known as a context free grammar. All of the grammars used in the CSL translator are context free grammars.

In order to express productions in BNF it is necessary to adopt some notational conventions. We will use alphameric identifiers to represent both nonterminal and terminal symbols. An identifier used in this context consists of a letter followed by zero or more letters or digits; the length of the identifier is arbitrary. Terminal symbols may also be represented by arbitrary strings of characters (including blanks); such strings will

be enclosed by a pair of $ symbols; e.g., $+ IS A PLUS SIGN$ is such a terminal symbol.

Productions may be written recursively; i.e., the production goal symbol may appear in the string on the right side. The grammar goal symbol will always be the production goal symbol of the first production listed for any grammar.

3. LR(k) PARSING

Parsing may be accomplished by two general methods. The first method, known as the top-down method, starts with the grammar goal symbol and proceeds to traverse the tree defined by the grammar down to the terminal symbols, looking for a match with the symbols of the sentence being parsed. This method is a trial and error process involving much backtracking and thus is very time consuming. Furthermore, detection of an error in the input sentence is virtually impossible until the traversal of the grammar tree has been completed; specification of the error location and subsequent recovery is quite difficult for large grammars such as those used by the CSL translator.

A far superior technique is the bottom-up method in which the parse starts with the terminal symbols of the input sentence and builds towards the grammar goal symbol. Many techniques have been devised for implementing a bottom-up parse. The technique used in the CSL Translator is the LR(k) method, which is capable of parsing a large subset of the deterministic context free grammars, such as the grammars used by the CSL Translator. The LR(k) method parses by making a single pass on the symbols of the input sentence from left to right, performing a reduction to a nonterminal whenever the rightmost leaves of the parse tree form a production. In the event of ambiguity in making the reduction by a particular production, the LR(k) method requires lookahead of at most k symbols to resolve the ambiguity.

The LR(k) parsing technique lends itself to the use of parsing tables to direct the parse. This permits a single parser to be used with many

different grammars simply by changing tables. Furthermore, it is feasible to generate these parsing tables automatically given a description of the grammar in BNF notation.

The LR(k) parser which is used in the CSL translator belongs to the class of machines known as deterministic push down automata. It can be thought of as a finite state machine coupled with a pushdown stack. The parser enters a sequence of states during the course of the parse, starting from an initial state. The parser makes a transition from its current state to a new state (called the destination state) depending upon either the next symbol read or the information on the top of the stack. Ultimately, the parser reaches a final state (called the accepting state) and the parsing action is complete.

In particular, there are three kinds of states which the parser can enter. They are known as read states, apply states, and lookahead states. Initially the parser begins by entering a specific read state with the stack empty. When it reaches the accepting state the stack will again be empty. The action which occurs in each type of state is detailed below.

When the parser enters a read state it immediately pushes the identity of the read state onto the stack. It then reads the next symbol in the input sentence. This symbol is obtained from the lexical analyzer in the form of a numerical code. The parser compares this symbol code with a prestored list of valid symbol codes which can occur at that point in the parse. Associated with each of the codes in the list is a corresponding destination state. If a match is found for the symbol code, the parser makes a transition to the appropriate destination state; if no match is found, an error exists in the input sentence, the parser announces this fact, and an error recovery routine is executed.

The parser enters an apply state whenever a BNF production in the grammar has been completed. That is, the top n items on the stack represent the n symbols which appear on the right hand side of a particular production (the items on the stack may represent both nonterminal and terminal symbols; the nonterminal symbols arise from the actions of previous apply states).

Of course, the stack actually contains read state identities which, in turn, represent the symbols of the production. The parser performs the reduction to the production goal symbol for the particular production which has been completed. This involves popping n - 1 read state identities off the stack, corresponding to replacing the n symbols on the right hand side of the production with the production goal symbol. The initial read state identity is not popped off the stack but is used instead to select the destination state for the parser. This remaining read state identity is known as the top state (because it is now on the top of the stack following the popping operation). The parser compares this top state with a prestored list of top states which are permissible at this point in the parse. Associated with each top state in the list is a destination state. The parser finds the match and gets the corresponding destination state. In actual practice, one of the permissible destination states is declared a default destination state; if a top state match is not found, then the default destination is automatically used. This does not jeopardize the error detecting ability of the parser in any way, since the parser is still deterministic and errors are discovered only by read states.

It is possible for ambiguities to arise at certain points in the parsing process. Such ambiguities require lookahead states to resolve the ambiguity. In the ambiguous situation the parser does not have sufficient information to select the correct destination state. One of the possible destination states is always an apply state. In addition, there will always be at least one other possible destination state which will be a read state. A lookahead state is entered to resolve the ambiguity. In the lookahead state the parser requests additional symbols from the lexical analyzer, and, on the basis of these lookahead symbols, determines the proper destination state. Thus, as a result of the lookahead inspection it may enter a read state or some apply state. In an LR(k) parser the maximum number of symbols which must be supplied in order to resolve all ambiguous situations is k symbols. For a particular lookahead state not all k symbols may be required (unless, of course, k = 1).

The symbols which are delivered by the lexical analyzer are not read by the parser in the lookahead state, but only inspected; in other words, the parser will request these same symbols again when entering subsequent read states. However, since the lexical analyzer delivers a particular symbol only once, the lookahead state queues these lookahead symbols within the parser on a first-in-first-out basis. Subsequent read states then obtain symbols from the queue instead of from the lexical analyzer until the queue is emptied. In this way the lexical analyzer is not required to backtrack its operation, just as the parser never backtracks.

The LR(k) parsing algorithm may now be stated (see Appendix O for structured programming conventions used in displaying algorithms):

```
*DESTINATION STATE = INITIAL READ STATE
*DO WHILE(DESTINATION STATE NOT ACCEPTING STATE)
   *IF(DESTINATION STATE = READ STATE)THEN
      *PUSH READ STATE IDENTITY ONTO STACK
      *READ NEXT SYMBOL
      *CHECK FOR SYMBOL MATCH
      *IF(SYMBOL MATCH FOUND)THEN
         *DESTINATION STATE = STATE ASSOCIATED WITH SYMBOL
      *ELSE
         *SYNTAX ERROR - PERFORM ERROR RECOVERY
   *ELSE
      *IF(DESTINATION STATE = APPLY STATE)THEN
         *POP READ STATE IDENTITIES OFF OF STACK
         *CHECK FOR TOP STATE MATCH
         *IF(MATCH FOUND)THEN
            *DESTINATION STATE = STATE ASSOCIATED WITH TOP STATE MATCH
         *ELSE
            *DESTINATION STATE = DEFAULT DESTINATION STATE
      *ELSE
         *DO SYMBOL LOOKAHEAD, CHOOSING DESTINATION STATE ASSOCIATED WITH
          MATCHING LOOKAHEAD SYMBOL STRING.  IF NO MATCH FOUND, DESTINATION
          STATE = LOOKAHEAD DEFAULT DESTINATION.
*STOP
```

The parser is completely deterministic; thus, if it receives a symbol from the lexical analyzer which is not a valid symbol at that point in the parse, the parser will announce an error and an error recovery process will be initiated. The error recovery process attempts to resume correct parser action with a minimum of loss of offending input text. Thus an LR(k) parser always detects errors as soon as they are encountered and attempts to recover from the error situation as quickly as possible so as to be able to detect additional errors.

Errors which are detected by the parser are known as syntax errors and are related to the grammatical structure of the sentence being parsed. In addition to syntax errors there are semantic errors which are detected by the semantic routines. The parser is incapable of detecting semantic errors because they are independent of and at a higher level than the grammar. Note also that the only source of syntax errors will be in the source language input text. Input sentences to all subsequent phases of the translation are generated by the first and subsequent phase semantic routines and are thus syntax error free. They may contain semantic errors, however.

Appendix A contains a detailed discussion of the LR(k) parsing method, including the method of generating the read, apply, and lookahead states for a grammar. Appendix B discusses the LR(k) Parser Generator, a program which automatically generates LR(k) parsing tables from a BNF description of a grammar.

4. LEXICAL ANALYSIS

Lexical analysis is performed by the lexical analyzer which can be conceptually regarded as being composed of a set of scanners. Whenever the parser requires an input symbol it calls upon the lexical analyzer to provide that symbol. In particular, the parser designates a specific scanner to provide the symbol. The reason for this is that generally it is not possible to construct a single scanner which can provide all the

symbols required by the parser during the entire course of the parse because of conflicts in the rules of combining characters to form symbols. Thus, at any given point in the parse it is required that all the possible symbols which the parser might legitimately receive be capable of being constructed by a particular scanner which the parser designates.

As mentioned earlier, there is really only a single scanner resident in the lexical analyzer. The multiple scanner concept is actually implemented by selecting an appropriate scanner table by which the behavior of the single scanner is programmed. Thus, we will speak of the scanner in this report as meaning the collection of scanners.

The scanner operates as a finite state machine with one character of lookahead. The scanner is composed of eight subscanners, any of which may be used to generate a symbol. The choice of subscanner is made on the basis of the first character used in generating the symbol. By means of a table lookup, indexed by the numerical code representing the first character, a choice of subscanner is made. The various subscanners have different properties, including the ability to transfer control to another subscanner, another scanner, or an external routine.

The lexical analyzer always returns a lexical code which identifies the lexical symbol to the parser. In addition, a semantic value may be returned for use by the semantic processing which is coupled to the parsing action. The semantic value may be a pointer to the character string which comprises the lexical symbol, the value of a numerical constant specified by the string, or any other arbitrary information.

The details of the lexical analyzer and a description of the Lexical Analyzer Generator which automatically generates lexical analyzer tables can be found in Appendix D.

5. SEMANTIC ROUTINE INTERFACE

The parser is only capable of recognizing grammatical constructs in the language; it has no ability to understand the meaning of the sentence being parsed. Thus, it is not capable of handling any part of the translation process directly and normally provides no output other than syntax error messages. On the other hand, translation and semantics depend upon a knowledge of the grammatical structure of the sentence. Therefore it is natural to use the parser to initiate various portions of the translation process.

This initiation is done by associating a semantic action with the recognition of a particular production in the grammar through the specification of a semantic code. This semantic code is written as part of the BNF notation describing the production and it selects a particular action within the semantic routine which may be called by the parser. The notation for semantic code specification requires insertion of the semantic code in parentheses immediately following the specification of the production goal symbol, for example:

MTERM(45) = MTERM AOP MFACTOR

If no semantic routine linkage is required the semantic code designation is omitted.

In order that the semantic routine has the necessary semantic information to work with, the parser pushdown stack actually contains information cells. These cells, in turn, contain parse, lexical, and semantic components. The parse component contains the read state history of the parse and is essential for the correct functioning of the parser. The lexical component contains the symbol codes which have been delivered by the lexical analyzer for each read state. The lexical analyzer also delivers semantic information along with the symbol code; this semantic information is contained in the semantic component. The semantic information is usually either a pointer or a value. For example, if an identifier is delivered, the semantic information consists of a pointer to the identifier character string; if a real number is delivered, the semantic information may be the value of the number in binary form.

Thus, it can be seen that the semantic component permits access by the semantic routine to all of the information known about the symbol. Furthermore, the cells on the pushdown stack are in direct correspondence with the terms of the production which has been recognized, with the cell on the top of the stack corresponding to the rightmost production term, etc. The parser is concerned only with the parse component (its interest in the lexical information occurs only at the time it receives the symbol code from the lexical analyzer). The semantic routine is primarily concerned with the information in the semantic component, and only moderately concerned with the lexical component; it is totally unconcerned with the parse component (in fact, it is incapable of interpreting the parse component information).

When the parser enters an apply state, it has recognized a production, represented by the stack contents. If a semantic code has been specified for the production the parser immediately calls the semantic routine. Using the semantic code as an index, the semantic routine performs whatever actions are needed. It may even alter the contents of the lexical and semantic components; but under no circumstances may it alter the parse component or the stack pointer. Following completion of the semantic action the parser pops the stack by changing the stack pointer, examines the parse component of the cell on top of the stack, and makes a transition to the appropriate destination state.

The semantic code = 1 is reserved for performing an initial semantic routine call. This call always occurs just before the parsing action begins. It permits any initial semantic actions to be performed. Also, since any production may call the semantic routine, the grammar goal symbol production provides a means of calling the semantic routine after all parsing has been completed. Thus, complete freedom exists in coupling the parsing action with semantic actions.

Generally, the semantic routines are concerned with generating a data structure which contains the input sentence for the next phase of the translation process. The semantic routine also provides for semantic error checking; for example, detection of a reference to a network element which has not been defined.

6. SYNTAX DIRECTED TRANSLATION

In addition to the general capabilities provided by the semantic routine, another mechanism is useful for translation to sentences composed of character strings. It is known as syntax directed translation. It has the virtue that it can be expressed directly in BNF notation, eliminating the need for any special routines to accomplish the required action.

In syntax directed translation (SDT) each production is represented by the ordered set (U, x, y) where U is the production goal symbol, x is a string of terminal and nonterminal symbols representing the definition of U, and y is a string of terminal and nonterminal symbols representing the translation of x. Since U is equivalent to x, U is also equivalent to the translation y of x. In BNF notation the ordered set is written in the form

U = x, y

In writing the string y we require that all nonterminals in y must also appear in x. The terminal symbols in y may be arbitrary. Furthermore, we associate a particular occurrence of a nonterminal in y with a particular occurrence of that nonterminal in x. If the nonterminals appear in the same order in x and y, this association is obvious since it is one-to-one. For example:

MTERM = MTERM $*$ MFACTOR, $*$ MTERM MFACTOR

Here we see that the first occurrence of MTERM in x corresponds to the first (and only) occurrence of MTERM in y, the first occurrence of MFACTOR in x corresponds to the first occurrence of MFACTOR in y, etc. Thus, the SDT consists of a mapping of nonterminals in one string to those in another string. There is no mapping permitted for the terminal symbols.

It is possible to specify mappings which are not one-to-one between x and y. This is done by assuming an implicit sequential numbering for each kind of nonterminal in x; y is then written with each nonterminal

suffixed with the number of the corresponding x nonterminal enclosed in parentheses. For example:

A = B F B G C B C, H C(2) B(3) B(1) H B(3)

where A, B, and C are nonterminals, and F, G, and H are terminals.

Note that particular nonterminals may be reused or omitted with this scheme. Thus, the first occurrence of C in y corresponds to the second occurrence of C in x, the first occurrence of B in y corresponds to the third occurrence of B in x, etc. The terminals F and G in x have no relation to the terminal H in y.

The syntax directed translation can be generalized to permit several syntax directed translations to proceed in parallel with the parsing. Furthermore, each SDT is permitted to access information belonging to another SDT.

In multiple SDT each production is represented by an ordered set (U, x, T1, T2, ..., Tn) where U is the production goal symbol, x is a string of terminal and nonterminal symbols representing the definition of U, and T1, T2, ..., Tn are translations of x. The ith translation is represented by the ordered pair (i, y_i) where y is a string of terminal and nonterminal symbols.

In BNF notation a multiple SDT is written in the form

$U = x,\ 1 = y_1,\ 2 = y_2,\ \ldots\ldots,\ n = y_n$

In writing the strings y we require that all nonterminals in y must appear in x. Again, the terminal symbols in y may be arbitrary. We associate a particular occurrence of a nonterminal in y with a particular occurrence of that nonterminal in x; in addition, we associate the semantic value of a particular nonterminal in y with a particular semantic value of the nonterminal in x.

Note that each nonterminal in x has n semantic values associated with it, one for each translation involved in generating that nonterminal. Thus,

the string y for each translation T may designate both the particular nonterminal occurrence in x and the particular semantic value attached to that nonterminal for purposes of constructing a particular translation. When the reduction to U is made, the n translations will correspond to the n semantic values to be attached to U, and thus the association is maintained.

We may append an ordered pair (j,i) to each nonterminal in the translation strings y, where j designates the nonterminal occurrence in x and i designates the semantic value of that nonterminal to be used in constructing y. If the nonterminals are referenced in the same order that they appear in x, the j index may be omitted by writing (,i); if the ith semantic value is used in the ith translation, the i index may be omitted by writing (j); and, finally, if both i and j can be omitted no ordered pair is required for designation.

An example of a multiple SDT is seen in the following example, where the production rule represents reduction of the product of a math expression term and factor to a term, the translation T1 represents the Polish prefix notation corresponding to the product, and translation T2 represents the Polish prefix notation for the total differential of the product:

```
MTERM = MTERM $*$ MFACTOR, 1 = $*$ MTERM MFACTOR, 2 = $+**$ MTERM (1,1)
   MFACTOR(1,2) MFACTOR(1,1) MTERM(1,2)
```

The most general form of SDT is obtained by also coupling a semantic routine with the SDT. This is notated by simply appending the semantic routine code number in parentheses to the production goal symbol as before.

When the parser enters an apply state characterized by a syntax directed translation, it first executes any semantic routine which may be designated. It then does the mapping of the specified SDT. Since the output of the SDT is a string, that string is constructed in another area of core and represented by a pointer. The reduction of the production to the production goal symbol is then made, popping the stack. Following the reduction, the semantic component of the cell on top of the stack contains the pointer to the string translation.

If a production does not require syntax directed translation it is written in the conventional manner without the y part.

SECTION VI

CSL TRANSLATOR STRUCTURE

The CSL Translator consists of two main phases. Phase 1 reads CSL source text, performs lexical analysis of the character string, and passes the lexical symbols to the LR(k) parser. The parser assembles the lexical symbols into productions which are specified by the BNF description of the CSL grammar. The parser calls specific semantic routines at particular points in the parse; these semantic routines assemble the canonic data structure which contains all of the essential information represented by the input text. The canonic data structure depends only on the semantic aspects of the source language. It thus represents the essence of the problem described by the input text without being encumbered by the syntactic properties of the language used to express that text. In fact, any equivalent language could be used instead of CSL to generate an identical canonic data structure.

The above actions comprise Phase 1 of the translator operation. These actions are independent of the target language, and the CDS is identical for all of the target languages for a given input text.

Phase 2 involves a set of translation steps involving selected portions of the CDS, followed by a syntax directed translation of the resultant data structure to the target language output text. The intermediate translations involve steps which are both dependent and independent of the particular target language. These steps will be discussed later.

The CSL Translator is constructed as an overlayed program, containing four overlays. The first overlay accomplishes Phase 1 in which CSL text is transformed into the CDS. The other three overlays represent the three target language translators, respectively, with a common segment used for target-language-independent translation steps and common routines.

The Phase 1 software must have the capability of coping with lexical, syntactic, and semantic errors. Lexical errors may occur because of improperly formed keywords, identifiers, or illegal characters. Syntactic errors may occur because of improper sequences of lexical symbols. Both of these errors are automatically handled by the lexical analyzer and the parser using error recovery embedded in the CSL BNF through user specification. Semantic errors may occur in Phase 1 due to meaningless or undefined items in the problem specification. These errors will be detected in the semantic routines which construct the CDS. If any of the three types of errors occurs during Phase 1 the program will halt at the conclusion of that phase. Thus, only error free input will actually be translated.

It is possible for semantic errors to be detected in Phase 2 also. Such errors are of two types: 1) additional errors caused by improper problem statement in the CSL text which are discovered by further semantic processing of the CDS, and 2) errors caused by entries in the CDS which cannot be expressed in the target language. An example of the latter type of error would be the specification of a transient optimization in CSL with an attempted translation to SCEPTRE; SCEPTRE cannot handle transient optimization. It is not possible for lexical and syntactic errors to occur during Phase 2.

1. MACRO SUBSTITUTIONS

The CSL language contains a macro substitution feature by which blocks of one or more logical cards may be defined and incorporated by reference into other parts of the text string. The definition of such blocks is accomplished by the TEXT directive, with reference made through the USE TEXT directive.

It is necessary to process these text blocks in the order in which they are referenced so that the syntactic and semantic analysis may be made in the context of their appropriate location in the input text structure. This permits meaningful diagnostics to be generated, clearly impossible if the text blocks are processed out of context, as shown by this example:

```
TEXT1
   PLOT, LINLOG, N(5)
   END TEXT
RUN
   USE TEXT1
   END
RUN MONTECARLO
   R6, GAUSS, .9
   USE TEXT1
   END
```

In this example TEXT1 represents legitimate CSL text except when used in the context of the RUN MONTECARLO entry (the keyword LINLOG is not permitted).

It is necessary that the definition of a text block be known at the point where that block is referenced. Therefore, in order to avoid a separate pass over the input text to locate all TEXT entries, it is necessary that all TEXT entries appear in the input stream before the corresponding USE TEXT directive.

When a text block definition is encountered, it is stored without syntax analysis in a special table. When the corresponding text reference is encountered, the character generator is switched to extract characters out of the text definition table instead of the normal input stream. In this way the text is expanded in context to the parser as though it had been in its proper position in the input stream. At the conclusion of the text definition the character generator is switched back to the input text stream. A text block may be referenced in this way as many times as desired.

Since text definitions may reference other text definitions it is possible for a set of nested references to be processed. The character generator is provided with a stack to aid in entering and leaving the various levels of the nested structure.

2. CANONIC DATA STRUCTURE

The CDS is built during the parsing of the CSL text stream by semantic routines which are called by the parser at key points in the parse. The CDS is organized as an n-ary tree. The nodes of the tree consist of data

blocks which are interconnected by pointers.

The root of the CDS tree is a node representing global data. This data consists of pointers to text definition blocks, model definition blocks, device definition blocks, the global network description block, the global level procedural block, and the solution blocks. This Global Block is the only block which is required in the CDS.

At any level in the CSL input there may be more than one entry of the same type. These multiple entries are linked together in the CDS through a pointer which occupies the first word of the CDS data block for that entry type. All other pointers in a CDS block access blocks which are subsidiary to the accessing block, that is, they are nested within that block. Each of the CDS blocks is given in detail below.

The Global Block has the structure:

1 Ptr to Subnet Definition Block chain for MODEL entries
2 Ptr to Device Block chain
3 Ptr to Subnet Definition Block chain for global level subnetwork definitions
4 Ptr to Procedural Data Block chain for global level
5 Ptr to Solution Block chain

The Subnet Definition Block occurs for each subnetwork or model definition. It has the structure:

1 Ptr to next Subnet Definition Block in current chain
2 Ptr to Subnet Header Block
3 Ptr to Datum Node Block chain
4 Ptr to Real Variable Definition Block chain
5 Ptr to Subnet Reference Block chain
6 Ptr to Subnet Definition Block chain
7 Ptr to Function Definition Block chain
8 Ptr to Table Definition Block chain
9 Ptr to Element Reference Block chain
10 Ptr to Procedure Definition Block chain
11 Ptr to Initial Conditions Definition Block chain
12 Ptr to Comment Block

The Device Block describes DEVICE entries and has the structure:

1 Ptr to next Device Block in current chain
2 Device name
3 Model name
4 Ptr to Value Block chain

The Procedural Data Block contains information associated with the problem solutions. It has the structure:

1 Ptr to Bound Block chain
2 Ptr to Run Controls Block chain
3 Ptr to Initial Conditions Block chain
4 Ptr to Curve Block chain

The Solution Block contains information relating to the types of calculation which may be specified in CSL. Solution Blocks are chained in the order in which the solutions are specified. The structure is:

1 Ptr to next Solution Block in current chain
2 Solution type
3 Ptr to Comment Block chain
4 Ptr to Parameter Block chain
5 Ptr to Swept Variable Block for swept variable
6 Ptr to Swept Variable Block chain for parametric variables
7 Ptr to Out Block chain
8 Ptr to Procedural Data Block chain
9 Ptr to Solution Block chain (for subsolutions as in optimization calculation)

The Subnet Header Block contains interface information for both subnetwork definitions and model definitions. The structure is:

1 Subnet name
2 No. of dummy nodes = n
3 No. of formal parameters = p
Dummy node names (n entries)
Formal parameter names (p entries)

The Datum Node Block has the following structure:

1 Ptr to next Datum Node Block in current chain
2 No. of nodes specified = n
Datum node names (n entries)

The Real Variable Definition Block specifies the real variables. The structure is:

1 Ptr to next Real Variable Definition Block in current chain
2 Real variable name
3 Ptr to lexical pair list

The lexical pair list contains the lexical and semantic code values for the lexical symbols which comprise the right hand side of the equation defining the real variable.

The Subnet Reference Block contains information for references to both subnetworks and models. It has the structure:

1 Ptr to next Subnet Reference Block in current chain
2 Subnet prefix
3 Subnet name
4 No. of nodes = n
5 Ptr to Value Block chain
Node names (n entries)

The Function Definition Block describes all user specified functions. It has the structure:

1 Ptr to the next Function Definition Block in current chain
2 Function name
3 No. of formal parameters = p
4 Ptr to lexical pair list
Formal parameter names (p entries)

The Table Definition Block has the structure:

1 Ptr to next Table Definition Block in current chain
2 Table name
3 No. of rows in table = r
4 No. of columns in table = c
Column values (c entries, present only if two-dimensional table)
Table data (ordering same as input sequence)

The Element Reference Block contains information for all standard elements which are referenced, using one block for each element. The structure is:

```
1  Ptr to next Element Reference Block in current chain
2  Element type
3  Element name
4  No. of nodes = n
5  Ptr to Value Block chain
   Node names (n entries)
```

The Procedure Definition Block specifies procedures. It has the structure:

```
1  Ptr to next Procedure Definition Block in current chain
2  Procedure name
3  No. of formal parameters = p
   Formal parameter names (p entries)
   Procedure text
```

The Initial Conditions Block has the structure:

```
1  Ptr to next Initial Conditions Block in current chain
2  Variable name
3  Variable modifier
4  Ptr to lexical pair list
```

The variable modifier specifies the type of initial condition associated with the variable, e.g., V(C1), Q(C1), or E(C1).

The Comment Block has the structure:

```
1  Ptr to next Comment Block in current chain
   Comment text
```

The Value Block is used to store value specifications. It has the structure:

```
1  Ptr to next Value Block in current chain
2  Value name
3  Ptr to lexical pair list
```

The Bound Block specifies information in the BOUNDS entry. It has the structure:

1 Ptr to next Bound Block in current chain
2 Parameter name
3 Ptr to Value Block for first specified value
4 Ptr to Value Block for second specified value
5 Shape name
6 LOG and relative value flags

The Shape name specifies the statistical distribution which is used, if any. The LOG and relative value flags are bit coded flags.

The Run Controls Block specifies one run control for each block. It has the structure:

1 Ptr to next Run Controls Block in current chain
2 Run Control name
3 Ptr to lexical pair list (optional)

The Curve Block specifies CURVE entry information and has the structure:

1 Ptr to next Curve Block in current chain
2 Curve name
3 No. of entries per row in curve data = n
  Curve entry data

The Parameter Block specifies modifications to be made to parameter values and has the structure:

1 Ptr to next Parameter Block in current chain
2 Parameter name
3 Ptr to lexical pair list

The Swept Variable Block is used to contain information for both swept and parametric variables and has the structure:

1 Ptr to next Swept Variable Block in current chain
2 Parameter name
3 Ptr to lexical pair list

The Out Block contains information about solution output displays and has the structure:

1 Ptr to next Out Block in current chain
2 Output type and flags
3 Ptr to Output Variable Block chain for dependent variables
4 Ptr to Output Variable Block for independent variable
5 Curve name
6 Weight

The Curve name and Weight are used in connection with objective function specification.

2

The Output Variable Block specifies the individual output quantities which may appear as output. The structure is:

1 Ptr to next Output Variable Block in current chain
2 Ptr to Comment Block for label text
3 Ptr to lexical pair list

3. TRANSFORMATIONS ON THE CANONIC DATA STRUCTURE

The CDS is generated by the initial semantic processing of the CSL input text. Inherent in the CDS structure is a hierarchical nesting of subnetworks, functions, procedures, tables, real variables, etc. In addition, there are micro substitutions permitted whereby character strings may be substituted through formal parameter specification in subnetworks and procedures.

The nesting capabilities of CSL far exceed the capabilities of any of the target languages. Also, the target languages do not have the capability of handling micro substitution. Therefore, it is necessary to denest the CSL description into a single level description, making micro substitutions as the denesting occurs, so that the final denested version is in a form which can be expressed using any of the target languages.

The denesting process is complicated by the fact that the nesting process may involve three kinds of replacements: terminal node name replacements, formal parameter replacements, and character string replacements.

These replacements are not necessarily simple replacements involving adjacent levels in the hierarchy, but may involve promotion of a name or string through many levels before the replacement is finally completed. The situation is further complicated by the fact that CSL employs the concept of "most local definition" in satisfying references. The "most local definition" concept locates definitions of referenced items by starting at the local level in the hierarchy; if no definition is found, then the next ancestral level is checked, and so on, back to the global level, if necessary, in order to find a definition. This process may also be carried to descendant levels in the hierarchy by using the nest notation, involving subnetwork names separated by dots. Thus, in CSL it is possible to reference any quantity in the hierarchical tree which is either encountered in the path back to the tree root or which is a member of the subtree belonging to the point in the hierarchical tree from which the reference is made.

It should be noted that the denesting and micro substitution process cannot be accomplished until the CDS or its equivalent has been established. That is, denesting can occur only when all of the hierarchical dependencies have been established through some internal data structure through which it is possible to traverse the hierarchical tree.

It is necessary to transform the CDS in the general case to an alternate structure in order to be able to accomplish the final translation to the target language. These transformations may be outlined briefly as follows:

1. All micro subsitutions must be made, regardless of target language. This may result in replication of some of the hierarchical entities. Replication requires internal renaming so that references can be made to the proper replicated entity.

2. Formal parameters for subnetworks must be eliminated for all languages. Again, replication and internal renaming may occur.

3. The "most local definition" problem must be addressed. This requires that all definitions be available on the level at which they are required for reference by the target language. This may involve replication of definitions and some renaming.

4. Only a single level of nesting may be permitted for SCEPTRE and CIRCUS-2. This involves expansion of the nest hierarchcy to eliminate higher level nesting for these languages.

The result of the above CDS transformations yields a new data structure which also conforms to the CDS format. Therefore, the denesting process simply transforms one CDS into another CDS which is more appropriate to the target language.

In practice it does not appear to be necessary to have the transformed CDS available in its entirety. Instead, the original CDS can be traversed in hierarchical fashion and a set of stacks can be maintained to handle the "most local definition" and other hierarchical problems. The secondary CDS can be generated "on the fly" with the final target translation occurring in synchronism with the creation and destruction of the dynamic secondary CDS. The final translation will utilize a mixture of semantic transformations and syntax directed translations coupled with the parse of the target language CDS.

Several details must be addressed during the denesting process. It is necessary to generate internal names because of replication, denesting, movement of definitions, etc. These names may be somewhat arbitrarily chosen for internal purposes, although they should be related to the nest notation used in CSL. However, such names are unsatisfactory as final names in the target language because of lexical rules for character content and length for indentifiers in the target languages. Thus, there are requirements for three naming conventions for an item: the original CSL name, the internal name associated with the denested version, and the target language name. Attention must be given to naming conflicts in the target language.

4. PUSH DOWN STACKS FOR DENESTING

A requirement exists for multiple pushdown stacks to facilitate the denesting process. These are required for associating terminal nodes, formal parameters, micro substitutions, and "most local definitions" with a particular level in the hierarchy.

A pointer is assigned as the primary linkage between a hierarchical entity and its associated push down list. The pointer accesses a linked list which is used to implement the push down stack. The top cell of the stack contains the most local information. The list is doubly threaded to permit easy popping of a hierarchical level off the stack. The list contains a pointer to a data cell which contains the actual data of interest. This data cell may be of arbitrary structure.

The push down stack cell is a 4-tuple with the following structure:

1 Ptr to next more global cell
2 Ptr to next cell on this hierarchical level
3 Ptr to data cell associated with this stack cell
4 Spare for arbitrary linkage

Let us show an example of the application of the push down stack. We will apply the stack to the "most local definition" problem associated with assigning values to circuit elements in nested BOUNDS entries. We have the following CSL statement of the problem:

```
BOUNDS
   R4   Data B
   R1   Data A
   R3   Data B
   END
RUN
   BOUNDS
      R1  Data C
      R2  Data D
      R4  Data C
   END RUN
```

Here we have BOUNDS information given for resistors R1, R2, R3, and R4. The BOUNDS data is specified by Data A, Data B, Data C, and Data D. The global BOUNDS information is superseded for R1 and R4 by a more local definition during execution of the RUN step. We will associate the global definition with level 1, and the RUN step definition with level 2. We will use the fourth word of the cell 4-tuple to store a pointer to the element description cell. In turn, each element description cell will contain a pointer to the push down stack for that element. The push down stacks are linked by level as well as from element to element for a given level. The BOUNDS data is stored in a set of separate data cells. A linked list header is established for each of the levels. A value of 0 is used to terminate a pointer chain.

Figure 4 shows the push down stack configuration when the RUN entry is active during denesting.

5. COUPLING OF THE CANONIC DATA STRUCTURE TO THE TARGET LANGUAGE

The CDS consists of a set of nodes which are organized into a tree structure. Each node contains linkages to other nodes of the tree as well as actual data. We require a means of intelligently traversing the CDS and generating a set of lexical symbols which can be used to construct the target language text. A simple blind traversal of the CDS is unsatisfactory because of the dissimilarity in the data structure of the individual nodes.

We construct a grammar using BNF notation which represents all of the "sentences" which can be formed by intelligent traversals of the CDS. We then perform the intelligent traverse (as described below), extracting lexical symbols from the CDS. These symbols are parsed by an LR parser, using the BNF tables for the CDS grammar.

Since we now have the capability of recognizing productions of the CDS grammar, we also have the capability of associating semantic translations and syntax directed translations with these productions. Thus, we are able to produce an output string representing the target text as a direct byproduct of parsing the CDS.

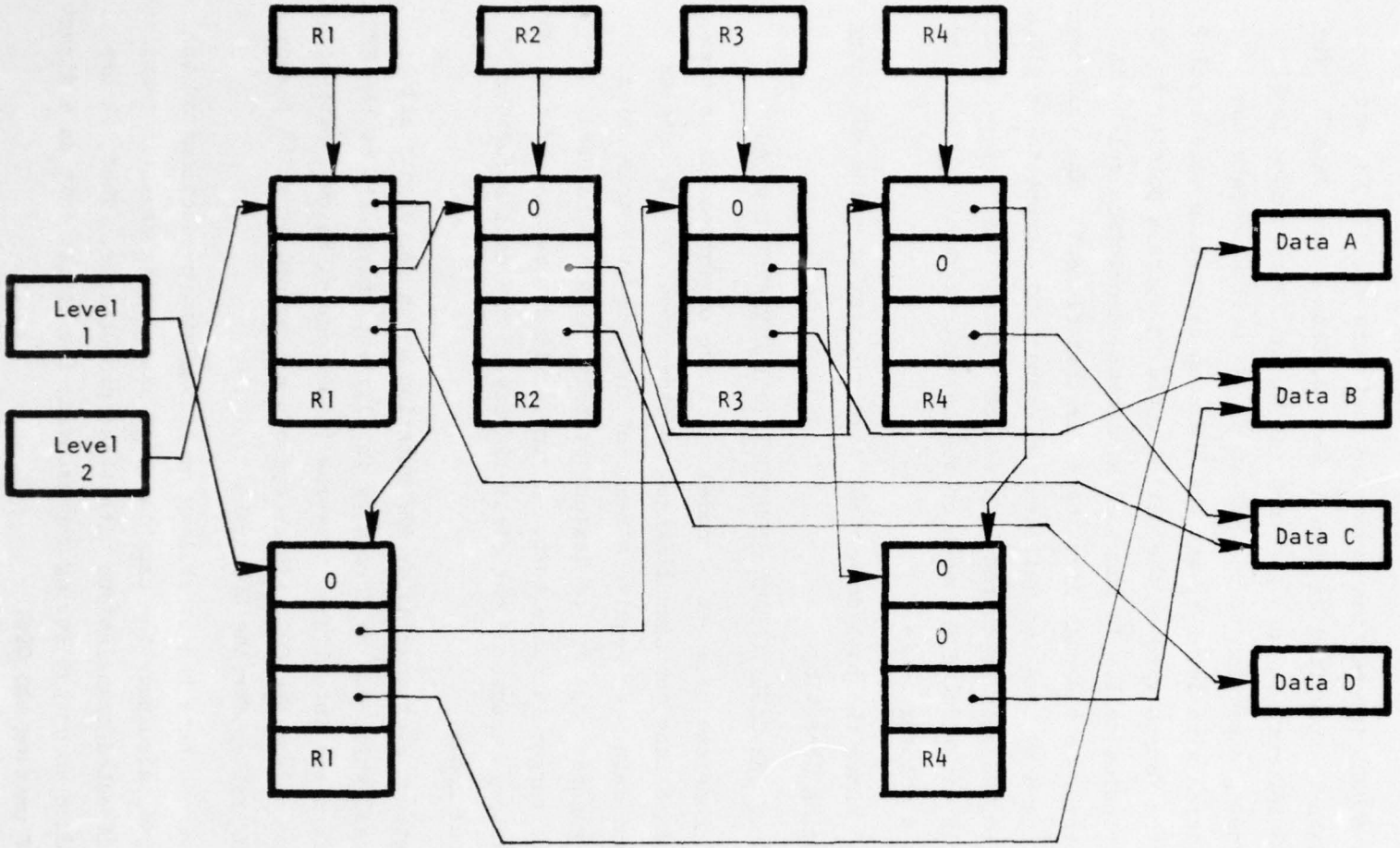


Figure 4. Example Push Down Stack For Denesting

The lexical analyzer and its component scanners are not required when parsing the CDS. Therefore, we specify scanner 0 in the lexical analyzer; this bypasses the normal lexical analysis and relies upon the character generator to produce the required lexical and semantic codes.

The character generator is permitted to perform arbitrary operations. It will obtain lexical and semantic symbols from two sources: 1) the lexical pair list for text strings which are available through the CDS, and 2) the semantic routines which have been previously executed in association with the parser action. These semantic routines will be capable of generating lexical and semantic symbols derived from inspection of the CDS. These symbols are queued for the character generator which subsequently delivers them back to the parser. The lexical symbols then control the course of the parse which in turn initiates other semantic routines which, in turn, generate additional lexical and semantic symbols. Thus the operation is bootstrapped, guided by the contents of the CDS through the semantic routines. It is highly desirable that the grammar for the CDS be LR(0) to avoid lookahead problems on the lexical symbols, since the lexical symbols are being generated just ahead of their use by the parser.

The purpose of employing the formal parsing technique in this phase of the translation is to permit the formal specification of the CDS grammar and the syntax directed translations. In this way, a formal coupling to the BNF of the target language is achieved.

Since the CDS consists of a finite number of node types it is desirable to assign particular semantic routines to processing particular CDS node types. Within the processing of a node there may be many steps; furthermore, the step sequence may be interrupted due to a traversal farther down into the CDS tree. We assign a set of states to the semantic routines which process each node type; these states indicate the status of processing for the node. A pushdown stack is maintained which stores the semantic routine number (and thus the node type) and the current processing state for that node when the processing is interrupted. Now we are able to

recover the processing status of a node as higher levels of the CDS tree are regained. This technique also permits recursive processing on a given node type.

From the above discussion it can be seen that the CDS-target language coupling mechanism depends upon two deterministic pushdown automatons operating in coroutine fashion, utilizing a BNF grammar description and the CDS for control of the operation. Auxiliary data structures can be generated as needed during the parsing operation. The modularity of the system permits very complex translations with very few basic translating routines.

SECTION VII

CONCLUSIONS AND RECOMMENDATIONS FOR SPECIFIC TRANSLATION DETAILS AND PROBLEMS

This section discusses CSL translation details and indicates several problem areas associated with translation.

The CSL Translator should meet two different goals. First, given a description of a problem in CSL, it is reasonable to expect that the translator can produce the corresponding target language text so that the same CSL problem description can be run on any of the three target language programs with correct results. Second, the CSL language and the translator should be comprehensive enough so that any problem which can be described in a particular target language can also be described in CSL.

The first goal can be met only if problems which utilize features common to all three target languages are submitted to the translator. This is not a very large subset of the problems which can be solved by using capabilities available in each of the programs used alone. The second goal can be met only by extending the CSL language and the translator to include capability for describing the various unique features of the individual target languages; this leads to redundancy in the language.

There are many features which are shared by the three target languages, but the techniques of describing problems using these features are sometimes vastly different. A simple example will make this point clear.

Let us take a current source whose value is described by a simple mathematical expression involving symbolic coefficients. The coefficient values are specified in other statements. The coefficients have been represented symbolically because we wish to change their values at a later time in the calculation. For this example we will use the diode equation (the fact that the diode equation is available in SCEPTRE will be overlooked, since we could have used another math expression which SCEPTRE does not have built into it). Let us describe this simple problem in the four languages:

In CSL:

```
J1,1-2=IS*(EXP(THETA*V(J1))-1)
IS=1E-9
THETA=30
```

In NET-2 (this will involve comp delay using Release 9 of NET-2):

```
I1  2  1  P1*(EXP(P2*V(I1))-1)
P1  1E-9
P2  30
```

In SCEPTRE (this will involve comp delay in SCEPTRE since we are not using the diode equation);

```
CIRCUIT DESCRIPTION
ELEMENTS
J1,1-2=X1(PIS*(EXP(PTHETA*VJ1)-1))
DEFINED PARAMETERS
PIS=1E-9
PTHETA=30
```

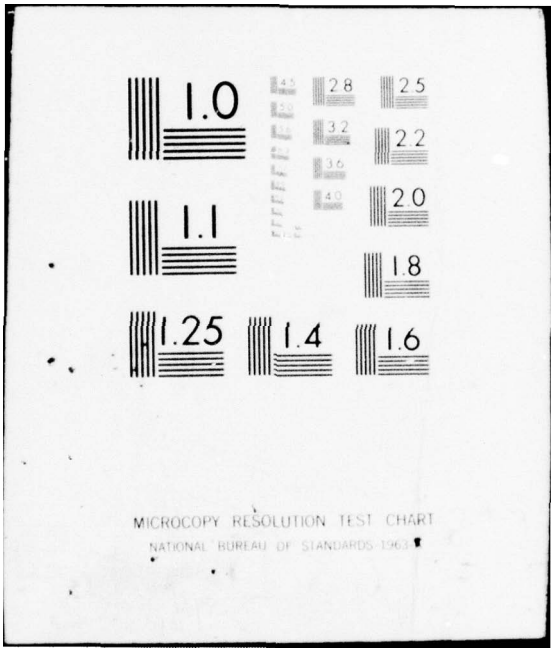
In CIRCUS (no comp delay involved):

```
MODELS
MODEL NAME=CURRENT GENERATOR
EXTERNAL NODES=(A,B)
TOPOLOGY
J1,A,B
EQUATIONS
       J1=IS*(EXP(THETA*V.J1)-1)
       D(J1,V.J1)=THETA*(J1+IS)
       RETURN
       END
END OF INPUT
DEVICES
DEVICE NAME = JGEN, MODEL NAME = CURRENT GENERATOR
SINGLE VALUED PARAMETERS
IS=1E-9
THETA=30
END OF INPUT
PREFIX,JG,CURRENT GENERATOR
JG1,1,2,JGEN
```

Note that in translating to CIRCUS that the translator must also determine the partial derivatives for the math expression and sequence the

AD-A033 296 BDM CORP EL PASO TEX F/G 9/2
ON THE DESIGN OF AN EXECUTIVE PROGRAM AND A COMMON SIMULATION L--ETC(U)
SEP 76 A F MALMBERG F29601-74-C-0017
UNCLASSIFIED BDM/E-47-75-F-0017 AFWL-TR-75-272 NL

2 OF 3
AD
A033296


1.0
1.1
1.25
1.4
1.6
1.8
2.0
2.2
2.5
2.8
3.2
3.6
4.0
MICROCOPY RESOLUTION TEST CHART
NATIONAL BUREAU OF STANDARDS 1963

equations in proper order in the EQUATIONS section. Thus, a translation to CIRCUS can certainly be specified, but implementing it is very complex. A mechanism for generating the partial derivatives for CIRCUS is discussed later in this section.

NET-2 contains many specialized elements which have no counterpart in the other languages. For example, in order to use the system analysis capability of NET-2 through the CSL language, it is necessary to have CSL elements available which permit access to all of the system elements in NET-2. However, translation of a problem using these elements in the other languages, although possible, is extremely difficult. In general, it involves having the translator write special subroutines and automatically interfacing these routines with the target simulation code in order to accomplish a satisfactory translation. It is felt that such an approach is outside of the scope of the translator project, although, from the user's point of view he may consider it a reasonable thing for the translator to do. Accordingly, a basic guideline has been adopted that the CSL Translator will not be concerned with modeling of elements.

1. SYNTAX DIRECTED TRANSLATION

Much of the target language text can be generated by means of a syntax directed translation (SDT). A SDT builds and concatenates output text strings by means of specified linear transformations involving symbols associated with the grammar during the parsing process. If semantic routine interfaces are permitted before executing a SDT for a given production, the semantic routine can supply any arbitrary or missing components which will be required by the SDT. In this way strings of any complexity can be constructed and placed in the output stream.

We will exhibit the application of SDT for a number of network elements below. The translation for each element will use the BNF notation for the original CSL format and the three SDTs for the target languages. The basic BNF and SDT notation is that used by the LR(k) Parser Generator described

in Appendix B of this report, except that the individual SDTs will be listed on separate lines according to the target language for each element.

For the resistor:

```
CSL:      RELEM = $R$ IDA $,$ NODE $-$ NODE $=$ VALUE
NET:      RELEM = $R$ IDA B NODE B NODE B VALUE
SCEPTRE: RELEM = $R$ IDA $,$ NODE $-$ NODE $=$ VALUE
CIRCUS:  RELEM = $R$ IDA $,$ NODE $,$ NODE $,$ VALUE
```

In the above, IDA is an alphameric string beginning with a digit and B is a blank. The translations for NODE and VALUE will have been provided by another production which defines these nonterminal symbols. From the above we see that the mapping requires no permutation of nonterminal symbols by any of the SDTs. The translation for capacitors is identical to that for resistors except that the terminal C replaces R.

For current sources:

```
CSL:      JELEM = $J$ IDA $,$ NODE $-$ NODE $=$ VALUE
NET:      JELEM = $I$ IDA B NODE(2) B NODE(1) B VALUE
SCEPTRE: JELEM = $J$ IDA $,$ NODE $-$ NODE $=$ VALUE
CIRCUS:  JELEM = $J$ IDA $,$ NODE $,$ NODE $,$ VALUE
```

Here we note that the node order is interchanged when translating from CSL to NET-2. Also, the element prefix is changed when translating to NET.

For voltage sources:

```
CSL:      VELEM = $V$ IDA $,$ NODE $-$ NODE $=$ VALUE $,$ VALUE
NET:      VELEM = $V$ IDA B NODE B NODE B VALUE $,$ VALUE
SCEPTRE: VELEM = $E$ IDA $,$ NODE(2) $-$ NODE(1) $=$ VALUE
CIRCUS:  VELEM = $V$ IDA $,$ NODE $,$ NODE $,$ VALUE
```

Here we note that CSL may specify a second value field representing series resistance. This resistance value is easily translated for NET-2, but cannot be handled by either SCEPTRE or CIRCUS-2 without inserting an extra series resistor in the network, thus introducing an additional node. On the other hand, if no series resistor is specified in CSL, this implies that the series resistance value is zero, creating numerical problems in the NET-2 solution.

This simple example illustrates some of the pitfalls encountered in automatic translation.

For the inductor:

```
CSL:      LELEM = $L$ IDA $,$ NODE $-$ NODE $=$ VALUE $,$ VALUE
NET:      LELEM = $L$ IDA B NODE B NODE B VALUE $,$ VALUE
SCEPTRE:  LELEM = $L$ IDA $,$ NODE $-$ NODE $=$ VALUE
CIRCUS:   LELEM = $L$ IDA $,$ NODE $,$ NODE $,$ VALUE
```

Here again there is a basis problem in handling the optional series resistor designation in CSL. The problem is identical to that for the voltage source.

For the mutual inductive coupling element:

```
CSL:      KELEM = $K$ IDA $,$ LNAME $-$ LNAME $=$ KVALUE
NET:      KELEM = $K$ IDA B LNAME B LNAME B KVALUE
SCEPTRE:  KELEM = $M$ IDA $,$ LNAME $-$ LNAME $=$ MVALUE
CIRCUS:   KELEM = $K$ IDA $,$ LNAME $,$ LNAME $,$ KVALUE1
```

Here we must convert the CSL value for coupling coefficient into the mutual inductance value for SCEPTRE using the mathematical transformation MVALUE = KVALUE/SQRT(L1*L2) where L1 and L2 are the values of the two inductors involved. The handling of this computation symbolically (as it must be handled by the translator) can be fairly involved if KVALUE, L1, and L2 are specified as math expressions. Also, for CIRCUS, KVALUE1 = - KVALUE.

This exhausts the circuit elements used in SCEPTRE except for source derivative terms, convolution kernels, and special math expressions. In the elements which follow we will be required to synthesize equivalent SCEPTRE elements, leading in some cases to very involved translations.

For the rational impedance/admittance element:

```
CSL:      ZYELEM = ZYPFX IDA $,$ NODE $-$ NODE $=$ NUMBER $,$ $($ NUM1 $)$ $,$
                   $($ NUM1 $)$
CIRCUS:   ZYELEM = ZYPFX IDA $,$ NODE $,$ NODE $,$ NUMBER $,$ $($ NUM1 $)$ $,$
                    $($ NUM1 $)$
```

This element is available only for translation to CIRCUS. The supporting BNF productions are:

```
ZYPFX = $Z$
ZYPFX = $Y$
NUM1 = NUMBER $,$ NUMBER
NUM1 = NUM1 $,$ NUMBER $,$ NUMBER
```

The pulsed voltage and current sources are CIRCUS elements but can be included in CSL for the convenience of CIRCUS users. Translation to CIRCUS is trivial but somewhat involved for the other languages:

```
CSL:    PVSOURCE = $PV$ IDA $,$ NODE $-$ NODE $=$ NUMFIELD EOLC
        PJSOURCE = $PJ$ IDA $,$ NODE $-$ NODE $=$ NUMFIELD EOLC

        NUMFIELD = NUMBER $,$ NUMBER $,$ NUMBER $,$ NUMBER $,$ NUMBER $,$
                   NUMBER $,$ NUMBER
```

where EOLC denotes End of Logical Card.

```
NET:    PVSOURCE = $V$ IDA B NODE B NODE B $TABLE$ IDA $(TIME)$ EOLC NUMLIST
        PJSOURCE = $I$ IDA B NODE(2) B NODE(1) B $TABLE$ IDA $(TIME)$ EOLC
                   NUMLIST

        NUMLIST = $TABLE$ IDA EOLC
                  T NUMBER(3) B NUMBER(1) EOLC
                  T NUM1 B NUMBER(2) EOLC
                  T NUM2 B NUMBER(2) EOLC
                  T NUM3 B NUMBER(1) EOLC
                  T NUM4 B $R$ EOLC
```

where NUM1, NUM2, NUM3, and NUM4 are given by arithmetic operations on the values represented by the lexical symbols:

```
NUM1 = NUMBER(3) + NUMBER(4)
NUM2 = NUM1 + NUMBER(5)
NUM3 = NUM2 + NUMBER(6)
NUM4 = NUMBER(3) + NUMBER(7)
```

and the lexical symbol T denotes an indentation level. The instances of the lexical symbol NUMBER refer to the ordering in the BNF production for NUMFIELD in the CSL grammar.

```
SCEPTRE:  PVSOURCE = $E$ IDA $,$ NODE(2) $-$ NODE(1) $=T$ IDA $(AMOD(AMAX(TIME-$
                     NUMBER(3) $,0)),$ NUMBER(7) $)$ EOLC
          PJSOURCE = $J$ IDA $,$ NODE $-$ NODE $=T$ IDA $(AMOD(AMAX(TIME-$
                     NUMBER(3) $,0)),$ NUMBER(7) $)$ EOLC
```

A string must be generated for later insertion into the FUNCTIONS section of SCEPTRE to define the table:

```
TABDEF = $T$ IDA $=$ NUMBER(3) $,$ NUMBER(1) $,$ NUM1 $,$ NUMBER(2) $,$
         NUM2 $,$ NUMBER(2) $,$ NUM3 $,$ NUMBER(1) $,$ NUM4 $,$
         NUMBER(1) EOLC
```

The lexical symbols NUM1, NUM2, NUM3, and NUM4 are derived through arithmetic operations in the same way as for the NET translation.

```
CIRCUS:  PVSOURCE = $PV$ IDA $,$ NODE $,$ NODE $,$ NUMFIELD EOLC
         PJSOURCE = $PJ$ IDA $,$ NODE $,$ NODE $,$ NUMFIELD EOLC
```

The definition for NUMFIELD for CIRCUS is identical to that for CSL. This SDT shows the need for semantic routine processing, dynamic string generation outside of the SDT, and the ability to use a nonterminal in several productions.

Sine wave sources are another CIRCUS feature which can be included as CSL elements. Translation to other languages is not as difficult as in the case of the pulsed sources:

```
CSL:        SVSOURCE = $SV$ IDA $,$ NODE $-$ NODE $=$ NUMFIELD EOLC
            SJSOURCE = $SJ$ IDA $,$ NODE $-$ NODE $=$ NUMFIELD EOLC
            NUMFIELD = NUMBER $,$ NUMBER $,$ NUMBER $,$ NUMBER

NET:        SVSOURCE = $V$ IDA B NODE B NODE B NUMFIELD EOLC
            SJSOURCE = $I$ IDA B NODE(2) B NODE(1) B NUMFIELD EOLC
            NUMFIELD = NUMBER(1) $+$ NUMBER(2) $*SIN(6.28318*AMAX(TIME-$
                       NUMBER(3) $,0)/$ NUMBER(4) $)$

SCEPTRE:    SVSOURCE = $E$ IDA $,$ NODE(2) $-$ NODE(1) $=$ NUMFIELD EOLC
            SJSOURCE = $J$ IDA $,$ NODE $-$ NODE $=$ NUMFIELD EOLC
            NUMFIELD = $X$ IDA $($ NUMBER(1) $+$ NUMBER(2)
                       $*SIN(6.28318*AMAX(TIME-$ NUMBER(3) $,0/$ NUMBER(4) $)$

CIRCUS:     SVSOURCE = $SV$ IDA $,$ NODE $,$ NODE $,$ NUMFIELD EOLC
            SJSOURCE = $SJ$ IDA $,$ NODE $,$ NODE $,$ NUMFIELD EOLC
            NUMFIELD = NUMBER $,$ NUMBER $,$ NUMBER $,$ NUMBER
```

CIRCUS also contains a tabular source element. Inclusion of this element in CSL and subsequent translation can be accomplished as follows:

```
CSL:      TVSOURCE = $TV$ IDA $,$ NODE $-$ NODE $=$ NUMFIELD EOLC
          TJSOURCE = $TJ$ IDA $,$ NODE $-$ NODE $=$ NUMFIELD EOLC
          NUMFIELD = NUMBER $,$ NUMBER
          NUMFIELD = NUMFIELD $,$ NUMBER $,$ NUMBER

NET:      TVSOURCE = $V$ IDA B NODE B NODE B $TABLE$ IDA $(TIME)$ EOLC
                     NUMFIELD
          TJSOURCE = $I$ IDA B NODE(2) B NODE(1) B $TABLE$ IDA $(TIME)$
                     EOLC NUMFIELD
          NUMFIELD = $TABLE$ IDA EOLC T NUMBER B NUMBER EOLC
          NUMFIELD = NUMFIELD T NUMBER B NUMBER EOLC

SCEPTRE:  TVSOURCE = $E$ IDA $,$ NODE(2) $-$ NODE(1) $=T$ IDA $(TIME)$
                     EOLC NUMFIELD EOLC
          TJSOURCE = $J$ IDA $,$ NODE $-$ NODE $=T$ IDA $(TIME)$ EOLC
                     NUMFIELD EOLC
          NUMFIELD = $T$ IDA $=$ NUMBER $,$ NUMBER
          NUMFIELD = NUMFIELD $,$ NUMBER $,$ NUMBER

CIRCUS:   TVSOURCE = $TV$ IDA $,$ NODE $,$ NODE $,($ NUMFIELD $),($ NUMFIELD
                     $)$ EOLC
          TJSOURCE = $TJ$ IDA $,$ NODE $,$ NODE $,($ NUMFIELD $),($ NUMFIELD
                     $)$ EOLC
          NUMFIELD = NUMBER
          NUMFIELD = NUMFIELD $,$ NUMBER
```

In the above SDT the first productions of NUMFIELD in each language are associated together, as are the second productions. In the CIRCUS translation there is a mapping of NUMBER into NUMFIELD which cannot be expressed solely through a finite SDT. Specifically, the nonterminal NUMFIELD appears twice in each of the CIRCUS productions for TVSOURCE and TJSOURCE. The appearances of NUMBER in CSL occur in pairs, with the first member of each pair assigned to the first occurrence of NUMFIELD in CIRCUS, and the second pair member assigned to the second occurrence of NUMFIELD in CIRCUS. This type of assignment is easily handled through a semantic routine.

2. PARTIAL DIFFERENTIATION OF MATH EXPRESSIONS

Partial differentiation of math expressions is required by CIRCUS for all math expressions entering into the network solution and by SCEPTRE for source elements under certain topological conditions. Therefore, the CSL Translator must have the ability of differentiating math expressions with

respect to specified variables. This differentiation can be accomplished by means of a syntax directed translation.

We will describe the SDT using the input notation for the LR(k) Parser Generator. In addition to the primary grammar, we have two SDTs, the first representing the original math expression, the second representing the derivative of the math expression. These translations will be labeled 1 and 2 for ease of identification. We will assume differentiation with respect to a particular identifier x. The SDT specification is:

```
S = E, 1 = E, 2 = E
E = AOP T, 1 = AOP T, 2 = AOP T
E = T, 1 = T, 2 = T
E = E AOP T, 1 = E AOP T, 2 = E AOP T
T = F, 1 = F, 2 = F
T = T $*$ F, 1 = T $*$ F, 2 = T(1,1) $*$ F(1,2) $+$ F(1,1) $*$ T(1,2)
T = T $/$ F, 1 = T $/$ F, 2 = $($ F(1,1) $*$ T(1,2) $-$ T(1,1) $*$ F(1,2)
                              $)/$ F(1,1) $**2$
F = F1, 1 = F1, 2 = F1
F = F1 $**$ F1, 1 = F1 $**$ F1, 2 = F1(1,1) $**$ F1(2,1) $*($ F1(2,1) $*$
                                    F1(1,2) $/$ F1(1,1) $+ALOG($ F1(1,1) $)*$
                                    F1(2,2) $)$
F1 = FUNC $($ ELIST $)$, 1 = FUNC $($ ELIST $)$, 2 = $($ Y $)$
F1 = i, T1 = i, T2 = W
F1 = $($ E $)$, 1 = $($ E $)$, 2 = $($ E $)$
ELIST = E, 1 = E, 2 = E
ELIST = ELIST $,$ E, 1 = ELIST $,$ E, 2 = ELIST E
```

In the above we note the symbol Y in the second translation for F1. The function reference represented by FUNC $($ ELIST $)$ is represented textually by a string of the form FUNC(A_1, A_2, ..., A_n). We may then define Y as the symbolic string representing the expression:

$$Y = \sum_i \frac{\partial Func}{\partial A_i} \frac{\partial A_i}{\partial E_i} + \frac{\partial Func}{\partial x}$$

where E_i is the ith E of ELIST in translation 2 and x is the differentiation variable.

The symbol W represents a value of one if identifier i is the differentiation variable x, otherwise W represents a value of zero. AOP represents the terminal symbols + and -, and i represents variable and numbers.

From the above we see that we can symbolically differentiate with respect to any independent variable using an SDT, except for a semantic routine intervention for the productions F1 = i and F1 = FUNC $($ ELIST $)$. We also require a differentiation with respect to each formal parameter of all functions which are referenced.

The translation containing the derivative may contain many superfluous one and zero terms as well as unnecessary parentheses. The one and zero terms arise from the translation of the production F1 = i. We now specify a SDT which accepts the derivative expressions as input and cleans up the superfluous ones, zeros, and parentheses pairs. In this specification we let MOP represent both * and / characters, and use U and Z to represent 1 and 0 values, respectively:

```
S = ES, ES
ES = E1, E1
ES = EZ, $0$
ES = EU, $1$
ES = E, E
E1 = T, T
E1 = EZ $+$ T, T
E1 = EU $+$ TU, $2$
E1 = E1 $+$ TZ, E1
EZ = TZ, NULL
EZ = $+$ TZ, NULL
EZ = EZ $+$ TZ, NULL
EU = TU, NULL
EU = $+$ TU, NULL
EU = EZ $+$ TU, NULL
EU = EU $+$ TZ, NULL
ELIST = ES, ES
ELIST = ELIST $,$ ES, ELIST $,$ ES
E = $+$ T, $+$ T
E = EU $+$ T, T $+1$
E = E1 $+$ TU, E1 $+1$
E = E1 $+$ T, E1 $+$ T
E = E $+$ TZ, E
E = E $+$ TU, E $+1$
E = E $+$ T, E $+$ T
T = F, F
T = T MOP F, T MOP F
T = T MOP FU, T
T = TU MOP F, F
TZ = FZ, NULL
```

```
TZ = T MOP FZ, NULL
TZ = TU MOP FZ, NULL
TZ = TZ MOP F, NULL
TZ = TZ MOP FZ, NULL
TZ = TZ MOP FU, NULL
TU = FU, NULL
TU = TU MOP FU, NULL
F = F1, F1
F = F1 $**$ F1, F1 $**$ F1
F = F1 $**$ FU1, F1
FZ = FZ1, NULL
FZ = FZ1 $**$ F1, NULL
FZ = FZ1 $**$ FU1, NULL
FU = FU1, NULL
FU = F1 $**$, NULL
FU = FZ1 $**$ FZ1, NULL
FU = FU1 $**$ F1, NU
FU = FU1 $**$ FZ1, L
FU = FU1 $**$ FU1, NULL
F1 = i, i
F1 = $($ E $)$, $($ E $)$
F1 = FUNC $($ ELIST $)$, FUNC $($ ENLIST $)$
F1 = $($ E1 $)$, E1
FZ1 = Z, NULL
FZ1 = $($ EZ $)$, NULL
FU1 = U, NULL
FU1 = $($ EU $)$, NULL
```

In the above SDT the symbol NULL denotes that there is no translation generated by the corresponding production. The above SDT does not include operations involving the arithmetic symbol -; extension to include this symbol is trivial.

Additional cleanup of differentiated expressions can be attained by arithmetically combining numerical constants and removing any superfluous parentheses which might result from that operation. An SDT is given below which accomplishes this. In this SDT terms may be represented either as lexical symbols or as numerical values. Semantic routines are required at certain steps to convert terms from symbolic to numerical form, from numerical to symbolic form, and to perform arithmetic operations on numerical forms yielding a numerical result. These steps are noted in the SDT which is given as:

```
S = SE, SE
SE = E, E
SE = EN, EN   (Note 1)
SE = E AOP EN, E AOP EN   (Note 1)
E = T, T
E = AOP T, AOP T
E = E AOP T, E AOP T
EN = TN, TN
EN = EN AOP TN,  EN AOP TN   (Note 2)
T = F, F
T = T MOP F, T MOP F
TN = FN, FN
TN = TN MOP FN, TN MOP FN    (Note 2)
F = F1, F1
F = F1 $**$ F1, F1 $**$ F1
FN = FN1, FN1
FN = FN1 $**$ FN1, FN1 $**$ FN1     (Note 2)
F1 = i, i
F1 = $($ E $)$, $($ E $)$
F1 = $($ E AOP EN $)$, $($ E AOP EN $)$   (Note 1)
F1 = FN1 $**$ F1, FN1 $**$ F1    (Notes 1, 4)
F1 = F1 $**$ FN1, F1 $**$ FN1    (Notes 1, 3, and 4)
F1 = FUNC $($ ELIST $)$, FUNC $($ ELIST $)$
FN1 = n, n
FN1 = $($ EN $)$, EN
ELIST = SE, SE
ELIST = ELIST $,$ SE, ELIST $,$ SE
```

Notes:
1. If numerical component present, convert to symbolic form.
2. If symbolic component present, convert to numerical form; perform indicated arithmetic operation and store result as translation in numerical form.
3. Handle exponent as an integer if possible.
4. Check for special cases and deliver appropriate results.

In the above SDT the symbol i represents non-numerical identifiers, while n represents numerical values.

3. PROBLEMS ASSOCIATED WITH EMBEDDED TARGET TEXT

It is desirable to have a means of embedding sections of target text in the CSL source text. This provides a mechanism for describing target

language features which are not available in CSL or are not handled by the translator.

Target text can be embedded through the ENTER entry. This entry specifies the target language in which the embedded text is written, followed by the text, and concludes with a special END card. For example:

```
ENTER NET
   R3  A  B  3.97*I(C5)
   C5  H  K  2.5
   END NET
```

When the CSL Translator is translating to a particular target language it will include all embedded target text which corresponds to the target language currently being addressed by the translator. Thus, embedded text can be included for more than one target language, with the appropriate text being used by the translator.

In most instances this feature cannot be used without specifying additional information. The problem arises in that the embedded text is not parsed by the translator, but is simply passed to the target text output stream. Thus, the identifiers which are used in the embedded text must correspond to the identifiers which will be used by the CSL Translator for target text which has been translated from CSL. Since these translator-chosen identifiers may be arbitrary names, a conflict arises. Consider, for example the following mixture of CSL and embedded NET text:

```
TN1,2-3=X
TN2,1-2=Y
SUBNET,TN,A-B=C
   R1,A-B=1/C
   ENTER NET
      TD1  3  B  1N604
      END NET
   END SUBNET
END
```

Here we see that the embedded text specifies node B. We assume that this is the same node B which is a dummy terminal node for subnet TN in CSL. The translator is required to denest this example since a formal parameter is being passed across the TN interface. Thus, node B becomes node 3 and node 2 in the denested circuit. However, since the embedded NET text is simply passed on to the output stream, there is no way of changing node B to the correct name. Alternatively, the embedded NET text could be parsed and translated; this alternative is highly unattractive since it requires complete semantic interpretation for all three target languages.

This problem can be alleviated in certain instances (but not for the example just given) by permitting the user to specify the names to be used in the target text. This is done by including an EQUIVALENCE entry on the global level in the CSL text. This entry has the format:

EQUIVALENCE, $s_1 = t_1$, $s_2 = t_2$, ... $s_n = t_n$

where s_1, etc. = the identifier name in CSL notation as referenced globally,
t_1, etc. = corresponding identifier name in target text to be used.

In this way the user can control the assignment of names. For example:

EQUIVALENCE, TN1.R1 = R101, TN2.R1 = R201

Problems still exist even with this feature. For example, suppose that R1 is both an element name and a node name in subnet TN; then what?

Another difficulty is that of association of the embedded target text with the correct location in the output text stream. For example, the relative location of the embedded text in the source text cannot be used reliably, since situations can arise where a single embedded text line must be associated with more than one location in the target text stream. For example:

```
BOUNDS
   ENTER NET
      R3 RECT .2  .6
      END NET
   R2 = .5, .8
   END
RUN MONTECARLO
RUN OPTIMIZATION
```

In translating to NET the embedded text line must be associated with both the Monte Carlo and optimization solutions. There appears to be no easy way of resolving the location association problem for embedded target text.

4. MISCELLANEOUS CONSIDERATIONS IN TRANSLATING CSL TO NET-2

Listed below are various miscellaneous details which must be considered when implementing a translation from CSL to NET-2. Many of the details are easily solved, others present more serious difficulties.

1. The indentation level structure of NET-2 must be created.

2. All comment cards must be moved to indentation level 0 in NET-2.

3. Datum node definitions require that translation to node 0 occur for the specified datum nodes. If a datum node is nested, dummy terminal nodes must be added to the subnetwork interfaces in the nest to provide a route back to the global node 0.

4. Order of node names must be permuted for certain elements.

5. Complex sources cannot be translated to NET-2.

6. Subnetwork definitions are translated to the NET-2 DEFINE entry, retaining as much information as possible. However, TABLE, CURVE, and BOUNDS entries are not allowed within DEFINE entries and must be removed.

7. NET-2 cannot handle procedures.

8. Initial conditions are built into certain element specifications in NET-2. Where applicable, they must be transferred from the CSL INITIAL CONDITIONS entry to the appropriate element description. If the initial conditions change they must be represented symbolically in NET-2.

9. Real variables which have a specified numerical constant value are translated to P constants.

10. Real variables which are defined symbolically are translated to X variables.

11. Element parameter symbols must be translated to sequence numbers for elements which are not handled as modeled devices in NET-2.

12. CSL function names must be converted to NET names by prefixing F1 to the name.

13. The sensitivity function is not available in NET-2.

14. The value modifiers PT, PV, and IV are not available in NET-2.

15. Time derivatives of real variables are translated to the corresponding X' variable.

16. Time integral values require a prescan of the CSL real variable equation system so that the appropriate real variables can be translated as X' variables. A conflict may exist in that the time derivative and time integral of a real variable may be required.

17. Math expressions may not be used for parameter value changes.

18. BOUNDS entry information must be incorporated inside of the Monte Carlo entry.

19. The CURVE entry must be translated to a DIST entry, ignoring any weights, for use by the Monte Carlo solution.

20. The nominal solution translates to a STATE entry. Run controls are not allowed.

21. Only small signal transfer functions of the form xM and xD (where x is A, B, Y, or Z) are allowed.

22. Output labels cannot be accommodated.

23. BOUNDS entry information must be incorporated within the optimization solution.

24. Math expressions are not allowed within the STATE entry except in the specification of an objective function.

25. Worst case solution is not allowed.

26. Out of bounds values in tables are handled by extension of boundary value in NET-2 as opposed to extension of boundary slope in CSL. Tables must be fixed up to allow reasonable out of bounds capability. This is non-trivial for two-dimensional tables.

5. MISCELLANEOUS CONSIDERATIONS IN TRANSLATING CSL TO SCEPTRE

Listed below are various miscellaneous details which must be considered when implementing a translation from CSL to SCEPTRE. Many of the details are easily solved, others present more serious difficulties.

1. All translated text must be organized under the SCEPTRE headings of MODEL DESCRIPTION, CIRCUIT DESCRIPTION, and RERUN DESCRIPTION.

2. Comments must be relocated to immediately follow a heading. Only 11 comment cards permitted for each heading.

3. Names must be restricted in length according to SCEPTRE rules. Care must be taken in choosing target names to insure uniqueness and retain meaning.

4. SCEPTRE does not have a global datum node. A node must be specified as the effective datum.

5. Only a limited set of electrical elements are available. Other CSL elements must be modeled; this can become very complex.

6. Source derivative expressions must be provided under certain topological conditions.

7. Small signal transfer functions are not permitted.

8. CURVE entry has no SCEPTRE counterpart.

9. Procedures must be converted to FORTRAN functions or subroutines.

10. Math expressions defining element values must be converted to SCEPTRE expression format.

11. Real variables translate to defined parameters and their time derivatives. A prescan is necessary to determine whether a real variable should become a defined parameter or the associated time derivative. Conflicts may arise due to the requirement for both the time integral and the time derivative of a real variable.

12. Functions must be converted to SCEPTRE equation format, with equation definitions placed under the FUNCTIONS subheading.

13. CSL names which occur in procedures and functions which are unacceptable to FORTRAN must be replaced and entered via the formal parameter list.

14. Two-dimensional tables are not permitted.

15. I and V are the only permitted time domain reponse variables.

16. Math expressions cannot be used to specify parameter value changes.

17. Problems occur in translating CSL parameter control information such as swept and parametric variable value specification. This includes control of time step and frequency intervals.

APPENDIX A

LR(k) PARSING

LR(k) parsing is a bottom-up parsing method in which input text is scanned from left to right (the L part), the rightmost sentential form is always reduced (the R part), and at most k symbols of lookahead are required to resolve ambiguities which may arise during the parse (the k part). LR(k) parsers are able to parse languages describable by LR(k) grammars, which include a large subset of the class of context free grammars. Virtually all computer languages in use today can be described by context free grammars (and, in some cases, by regular grammars, which are special cases of the context free grammars). In addition, LR(k) parsers process text in a time proportional to the length of the text string and never backtrack. Practical parsers are deterministic and are guaranteed to detect and announce syntax errors immediately upon encountering them.

From the above remarks it can be seen that LR(k) parsing methods have many advantages and virtually no disadvantages compared to other parsing methods. In the past, LR(k) parsers have had a serious disadvantage in that the tables required to direct the parse were extremely large, even for a small grammar. Recent advances in implementing LR(k) parsers have lead to a substantial reduction in table size, much of the reduction being due to packing and optimization techniques. The LR(k) parser which has been developed for the CSL Translator takes advantage of these techniques.

Basically, an LR(k) parser operates by scanning lexical symbols representing the input text and storing a history of the parse in a pushdown stack. Thus, it operates as a finite pushdown automaton. By means of a lookahead capability it is possible to make the parse completely deterministic at every stage.

It is useful to develop the LR(k) parser in terms of a finite state machine without a pushdown stack initially. We will then add the pushdown stack which will permit the LR(0) languages to be parsed. Finally, the lookahead capability will be added permitting the LR(k) languages to be parsed.

1. THE CHARACTERISTIC FINITE STATE MACHINE

We will begin the development of an LR(k) parser by building a characteristic finite state machine (CFSM). We assume that any sentence in the language which is to be parsed is always terminated by a special end symbol $. The grammar may contain left and right recursive productions. We choose for purposes of illustration the following grammar which describes simple mathematical expressions:

```
1  S = E $
2  E = T
3  E = E + T
4  T = F
5  T = T * F
6  F = i
7  F = (E)
```

where the terminal symbols are $, +, *, i, (, and). We number the productions with the integers 1 through 7. We will also number the symbols on the right hand side of each production from left to right, starting with the number 1. We further assume that each production has a special terminal symbol # appended to its right end.

We can specify an ordered pair m,n which permits us to designate a particular term in production m, term n. Thus, term 1,2 is the symbol $.

Let us now establish the CFSM by creating an initial state, S1. We will initialize S1 with all first terms of all productions which define the grammar goal symbol (in this case term 1,1 since the grammar goal symbol S is defined only by production 1). We can specify a CFSM state and the production terms of which it is composed by listing the CFSM state number, the production and term number, and the grammar symbol

which corresponds to that production and term. Thus we have initialized the state S1 as follows:

S1 1,1 E

The information which we have listed for S1 is called the basic state of S1. The basic state contains sufficient information to show us exactly where we are at some given point in the parse. If the basic state contains only terminal symbols, then the basic state is the complete CFSM state. However, if the basic state contains nonterminal symbols, as is the case in S1 in the example, then each nonterminal symbol represents one or more productions which can be added to the CFSM state, with each new production represented by the first symbol of that production. Again, if a new nonterminal is introduced into the CFSM state, the state is again augmented by the first symbol of all productions defined by that nonterminal. This process continues until no new productions can be added to the state, at which time the state is complete. Thus, expanding the basic state for S1 we find the complete state for S1 to be:

S1	1,1	E
	2,1	T
	3,1	E
	4,1	F
	5,1	T
	6,1	i
	7,1	(

The underline is used to delineate the basic state from the complete state. We note that the details of expansion for the above state are as follows:

1. The basic state is 1,1 E.
2. E is a nonterminal and introduces productions 2 and 3. We therefore augment S1 with terms 2,1 T and 3,1 E.
3. 2,1 T is a nonterminal represented by productions 4 and 5. We therefore augment S1 with terms 4,1 F and 5,1 T.

4. 3,1 E is a nonterminal, but all productions for E have already been included in S1, therefore there is no further action associated with this term.
5. 4,1 F is a nonterminal represented by productions 6 and 7. We therefore augment S1 with terms 6,1 i and 7,1 (.
6. 5,1 T is a nonterminal which has previously been accounted for, therefore there is no further action associated with this term.
7. 6,1 i is a terminal symbol, therefore no further action is associated with this term.
8. 7,1 (is a terminal symbol, therefore no further action is associated with this term.

We note that the state S1 shows the terms in the grammar which are currently involved in the reading of the terminal symbols contained in the state. That is, we will read either an i or (symbol at this stage of the parse; this in turn means we are in the process of forming the nonterminals represented by 1,1; 2,1; 3,1; 4,1; and 5,1. Also, since this is the initial state, we are in the process of initiating the grammar goal production. Thus, the CFSM state specifies exactly where we are in the parsing process.

Each nonterminal or terminal symbol except # in a state is followed by a subsequent symbol which is a member of another state, called the destination state for the nonterminal or terminal symbol. That is, when a terminal symbol has been read and recognized, or a nonterminal construct has been completed, a transition to a new CFSM state is made. This permits us to define the next CFSM state S2. We will assign S2 as the destination state for the nonterminal E in state S1. That is, whenever we have completed the construction of E in S1 we will then enter state S2. Accordingly, we speak of S2 as the destination state of E in state S1 and enter the destination state in the destination state column for E in S1. The first entry in S1 now reads:

S1 1,1 E S2

The basic state of the destination state can now be defined. It contains the successor terms of the terms which caused a transition to the destination state. Thus, in the example, the basic state for S2 contains 1,2 (the successor to 1,1) and 3,2 (the successor to 3,1). However, if the new destination state basic state is identical with a basic state of a previously developed state, then the transition is always made to the previously developed state. In other words, the basic states define the unique configurations of the grammar which can occur during parsing; a duplication of a basic state would destroy that uniqueness.

The end of a production is signaled by the appearance of the # symbol as a term. In this case there is no expansion to other productions. Rather it signifies that a production has been completed and that a reduction to the production goal symbol is to occur. Thus, there is no destination state associated with the # symbol.

When all nonterminal transitions have been completed and no new CFSM states have been generated the CFSM is complete. The complete CFSM for the example grammar is shown below:

S1	1,1	E	S2
	2,1	T	S3
	3,1	E	S2
	4,1	F	S4
	5,1	T	S3
	6,1	i	S5
	7,1	(	S6
S2	1,2	$	S7
	3,2	+	S8
S3	2,2	#	
	5,2	*	S9
S4	4,2	#	
S5	6,2	#	
S6	7,2	E	S10
	2,1	T	S3
	3,1	E	S10
	4,1	F	S4
	5,1	T	S3
	6,1	i	S5
	7,1	(	S6

S7	1,3	#	
S8	3,3	T	S11
	4,1	F	S4
	5,1	T	S11
	6,1	i	S5
	7,1	(	S6
S9	5,3	F	S12
	6,1	i	S5
	7,1	(	S6
S10	7,3	)	S13
	3,2	+	S8
S11	3,4	#	
	5,2	*	S9
S12	5,4	#	
S13	7,4	#	

Although the CFSM enumerates the various configurations of the grammar which may be encountered during the course of parsing we note that a practical parser cannot be constructed using a CFSM expansion for the example grammar for two reasons. First, we note that in S3 we must decide between reading a * symbol and applying production 2. Similar problems occur in S11. Second, there is not yet any formal mechanism for knowing the destination state following the application of a production. The first problem is solved by the lookahead process; the second by adding a pushdown stack to create a pushdown automaton.

2. THE DETERMINISTIC PUSHDOWN AUTOMATON

It is necessary to convert the CFSM into a deterministic pushdown automaton (DPDA) in order to be able to unambiguously parse LR(k) grammars. We will describe the process in two steps. The first step will detail the pushdown automaton. The second step will describe the conversion to a deterministic pushdown automaton.

We note from our example grammar that we require a means of determining the destination state for our automaton when a production is reduced to its goal symbol. The process of reducing a production to its goal symbol is called the application of the production. Clearly the destination state is known if we know where the production being applied was initiated in the CFSM, since then the destination would be the destination state for the production goal symbol at that point in the CFSM.

We introduce a pushdown stack to save the detailed history of the parse. If we save the CFSM state which is entered at each step of the parse, then clearly we have a detailed history of the parse and can determine in principle where to make a transition in the event of application of a production.

We now describe the basic operations which our pushdown automaton can perform. We can read a terminal symbol, in which case the symbol is consumed and the input text pointer is advanced to the next symbol; this is called a read operation. We can apply a production, in which case no input is read but instead a sequence of one or more grammar symbols is reduced to a single production goal symbol. We can enter an accepting state which signals the successful completion of parsing of the input text by a final reduction to the grammar goal symbol. Finally, we can enter an error state because the input symbol which has just been read is unacceptable to the grammar at that point in the parse.

The CFSM states can be divided into three kinds of states: read states, apply states, and inadequate states. CFSM states which contain only transitions on terminal and nonterminal symbols are called read states. CFSM states which contain only a single application of a production are apply states. All other CFSM states are inadequate states. The CFSM becomes nondeterministic whenever an inadequate state is entered; that is, it is not known whether a new symbol should be read or a production application should be made. The nondeterministic CFSM states can

be converted into deterministic states by means of a lookahead process which determines the correct action on the basis of symbols yet to be read.

We note that input symbols are consumed only by read states. Since all possible symbols which can occur at a given point in the parse are included in the terminal symbol set in a particular read state, we see that the parser has the ability to announce an error if any symbol other than those in the read state terminal symbol set is encountered. Thus, the parser has the ability of detecting a syntax error immediately upon encountering an incorrect symbol.

We also note that we can save the history of the parse on the pushdown stack by pushing each read state identity onto the stack as the read states are entered. We will include the read transitions of an inadequate state as a read state associated with that inadequate state so that the stack may also contain this read state identity.

Let us consider a production which contains only terminal symbols. We note that each terminal symbol will be read by a particular read state. When the # term of the production is encountered, signifying that the production is to be applied, the stack will have been augmented by a number of read state identities equal to the number of terms in the production. When the production is applied the stack is popped, removing all read state identities except that corresponding to the first term of the production. This read state identity remains on the stack and represents the nonterminal symbol to which the production is being reduced. As the parsing continues this stack entry represents the nonterminal term in some production which initiated the production just applied.

From the above it can be seen that the stack always contains entries representing terms of partially completed productions (except for the top of the stack which may contain a complete production about to be applied). Furthermore, these uncompleted productions extend back to the very start of the parsing operation. The reduction of the top set of entries to a nonterminal represents the reduction of the rightmost sentential form in the LR(k) parsing process.

The pushdown stack provides the means by which the parser knows what destination state to choose following the application of a production. In developing the CFSM states we keep track of the states in which each production is initiated. The set of initiating states for a production are known as the top states for that production. There will always be a read state on the stack corresponding to an initiating state for a production. When the production is applied, the stack is popped, removing all stack entries except that corresponding to the read state which initiated the production; hence the name top state, since the initiating read state is now on the top of the stack.

Since the production which has been applied is known, we know what nonterminal has been generated by the production application. Since we also know the top state which initiated the expansion of that nonterminal (i.e., initiated the production just applied) we know the destination state for that nonterminal. Thus, when a production is applied we know immediately from the top state identity what the resulting nonterminal destination state is. This permits the nonterminals to be completely removed from the parsing table, resulting in substantial savings in table size.

We are now in a position to describe the DPDA states. They are generated from the CFSM states as follows: The terminal symbol read transitions of a CFSM read state comprise a corresponding DPDA read state. The terminal read transitions of a CFSM inadequate state comprise a corresponding DPDA read state. Each production application in a CFSM apply or inadequate state becomes a DPDA apply state. CFSM inadequate states become DPDA lookahead states. The initiating states for each production in the CFSM states become top state identities for that production in the DPDA.

Let us illustrate these concepts using our example grammar. We will label the DPDA states with the letters R, A, and L, corresponding to read states, apply states, and lookahead states. We will retain the same numbering for read and lookahead states as was used in the CFSM;

however, we will label apply states with the original production numbering scheme.

The read states for the example grammar are:

```
R1   i   A6
     (   R6

R2   $   A1
     +   R8

R3   *   R9

R6   i   A6
     (   R6

R8   i   A6
     (   R6

R9   i   A6
     (   R6

R10  )   A7
     +   R8

R11  *   R9
```

The apply states may be specified by listing, in order, the apply state identity using the original production numbers, the number of entries to be popped from the stack (one less than the number of terms in the production), and the top state vector identity. There is a top state vector for each unique nonterminal, representing one or more productions which reduce to that nonterminal. The top state vector for a nonterminal contains the DPDA read state identities corresponding to the initial read state for the corresponding production set. The top state vector numbering uses the original production numbering for the first production which defines a

particular nonterminal. Top state identities are prefixed with a T. The DPDA apply states are:

```
A1  1  T1
A2  0  T2
A3  2  T2
A4  0  T4
A5  2  T4
A6  0  T6
A7  2  T6
```

The top state vectors list, in order, the top state identity, the DPDA read state which may occur as a top state, and the destination state corresponding to that top state. For the example grammar we have:

```
T1  R1  0

T2  R1  R2
    R6  R10

T4  R1  L3
    R6  L3
    R8  L11

T6  R1  A4
    R6  A4
    R8  A4
    R9  A5
```

We note that nonterminal S is defined in production 1, therefore we will have a top state T1; similarly, nonterminal E is defined in productions 2 and 3, therefore we will have a top state T2 since that is the first production defining E. Next we note that the production covered by T1 (production 1) is initiated in CFSM state S1, with a corresponding DPDA read state R1; therefore, T1 has R1 as a top state. Similarly, the productions covered by T4 (productions 4 and 5) are initiated in CFSM states S1, S6, and S8, with corresponding DPDA read states R1, R6, and R8; therefore, T4 has top states R1, R6, and R8.

Next let us inspect the destination states in the top state vector. The destination state associated with top state R1 in top state vector T1 is special; if that production is applied then we have reached the grammar goal symbol and the parsing is complete; thus a special destination state 0 is reached which signals the satisfactory completion of the parsing operation. If we use top state vector T2 we know that we have just applied either production 2 or 3, resulting in nonterminal E. If the initiating state for the production was CFSM S1 (corresponding to DPDA state R1), we can see that the destination associated with E in S1 is S2, which corresponds to a DPDA destination state R2. Thus, in T1 a top state R1 leads to a destination state R2. In T4 we have applied either production 4 or 5, resulting in nonterminal T. If the top state is R8 (corresponding to CFSM state S8), we see that a transition on T in S8 takes us to state S11 which is an inadequate state. State S11 corresponds to DPDA lookahead state L11. Thus, in T4 a top state R8 takes us to a destination state L11.

3. LOOKAHEAD STATES

The DPDA will contain lookahead states whenever the CFSM contains inadequate states. A lookahead state must examine the next n symbols ($n \geq 1$) in order to decide whether to enter a read state or apply a production. The amount of lookahead required by each lookahead state may be different, but the greatest amount of lookahead required by any lookahead state in an LR(k) grammar is k symbols. Thus, if the grammar is LR(0) no lookahead is required and the DPDA as currently described is adequate for parsing such a grammar. Generally, grammars do not tend to be more complex than LR(3).

The basic problem in defining a lookahead state is to determine the minimal sequence of lookahead symbols which must be examined in order to resolve the ambiguity. On the basis of the lookahead string a transition can then be made either to an apply state or a read state. The lookahead algorithm to be described here is applicable only to the parsing of the grammar itself. When error recovery and semantic actions internal to the production are required, the lookahead process becomes more complex.

Basically, the lookahead algorithm consists of a parallel expansion of the possible transition paths, inspecting as many terminal symbols as may be required to resolve the lookahead ambiguity. When the parsing tables are being generated there is no specific input symbol string; thus it is necessary to consider all possible strings when developing the lookahead tables.

Each lookahead state can be initially separated into a set of basic substates. Each basic substate is either a read state, containing all terminal read symbols, or an apply state containing a single apply (in fact, the basic substates are identical with the states which are immediately entered upon resolution of the lookahead problem). Each of the basic substates is then expanded until the next terminal symbol is read in each expansion. At this point the terminal symbol string associated with each basic substate is examined for uniqueness. Those basic substates which have unique strings have been resolved with regard to lookahead; those basic substates which do not have unique lookahead strings must be extended further until another terminal symbol has been read and added to the string. This process is continued until there is a unique terminal string associated with each basic substate. The amount of lookahead associated with a particular lookahead state is then the length of the longest string required by the algorithm to separate its basic substates.

In performing the expansion the following rules are used: For read states all terminal symbol transitions are employed. All top state transitions are used for apply states. Any lookahead state which is encountered must be separated into basic substates and they in turn are expanded. Thus, the expansion of a lookahead state may invoke additional lookahead states; the separation of secondary lookahead states into their basic substates does not require unique strings for these substates however.

The lookahead expansion is best visualized as a graph in which the root of the graph is the original lookahead state, the immediate descendant nodes of the root represent the substates, descendants of the substate nodes represent various DPDA states which are encountered during the expansion, and the branches represent transitions from state to state. Branches which correspond to read transitions on terminal symbols are labeled with the terminal symbol. As soon as a basic substate of the original lookahead state has a unique terminal symbol string attached to it, it has been separated from the lookahead expansion and is removed from further consideration. The expansion is done using the DPDA information only.

The lookahead expansion using the graph representation for the lookahead states L3 and L11 of the example grammar is shown in Figure A1.

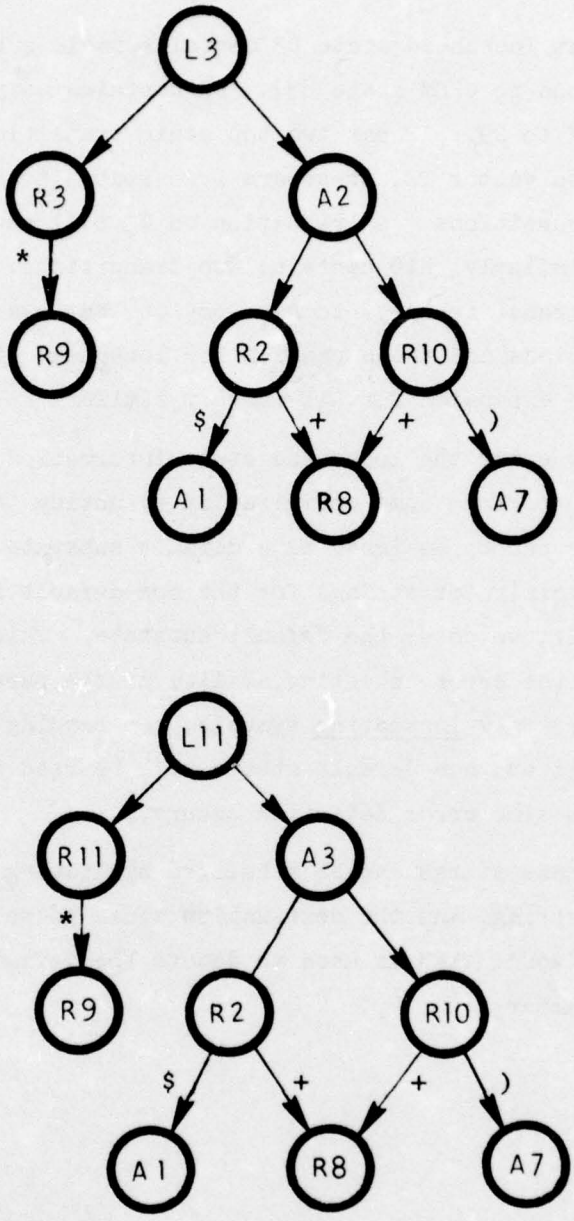


Figure A1. Lookahead Tree Expansion for Example Grammar

We see that lookahead state L3 contains basic substates R3 and A2 (this corresponds to CFSM state S3). R3 contains only a single read transition on * to R9. A2 has two top state transitions in its associated top state vector T2; these are transitions to R2 and R10. R2 contains two transitions: a transition on $ to A1 and a transition on + to R8. Similarly, R10 contains two transitions: a transition on + to R8 and a transition on) to A7. Now the basic substates R3 and A2 have unique strings and the expansion for lookahead state L3 has been completed. The expansion for L11 is very similar.

We can now enter the lookahead state information into the parsing tables. We can perform some optimization by noting that one of the basic substates can be assigned as a default substate. Then we only need test explicitly for strings for the non-default substates, and, if they all fail, we enter the default substate. This approach does not compromise the error detecting ability of the parser since during lookahead we are only inspecting symbols, not reading them. The symbols for both default and non-default states will be read in subsequent read states at which time error detection occurs.

The lookahead states can be tabulated by listing the state identity, the lookahead string, and the destination state to be taken if the string is found. The word OTHER is used to denote the default condition. For the example grammar:

```
L3  *      R3
    OTHER  A2

L11 *      R11
    OTHER  A3
```

This completes the discussion of the derivation of the LR(k) parsing tables under the condition that no error recovery or semantic actions are permitted within a production. The extension to these situations will be covered later.

4. LR(k) PARSER OPERATION

The LR(k) parser operates as a finite state deterministic pushdown automaton, reading symbols provided by the lexical analyzer, pushing read states onto the stack, and popping states off the state when productions are applied. Lexical symbols are consumed only by read states; lookahead states examine lexical symbols but do not consume them --- they are consumed by subsequent read states.

The parser begins with the stack empty and automatically transitions to an initial state. In the absence of error recovery and semantic actions internal to the productions the initial state will always be the read state R1. Thus, when the initial transition to R1 is made, the parser will push R1 onto the stack. Since a read state has been entered, the lexical analyzer will be called to deliver the first lexical symbol. This symbol is compared against terminal symbols in the parsing table for the read state. If comparison is found a transition is made to the specified destination state. If no comparison is found, the parser signals a syntax error; if error recovery capability has been included in the grammar, the error recovery mechanism will be initiated.

Subsequent states may be read, apply, or lookahead states. Whenever an apply state is entered, the stack is popped, the stack top state is examined, and a transition is made to a destination state specified by the top state. When a lookahead state is entered the parser requests symbols from the lexical analyzer in a lookahead mode and traverses the lookahead symbol strings stored in the parser tables, comparing the actual symbols against the strings in the tables. When sufficient symbols have been found to determine the lookahead destination state, the parser leaves the lookahead mode and makes a transition to the appropriate state. The symbols delivered by the lexical analyzer in the lookahead mode will be redelivered to the parser in subsequent read states, although lexical analysis will not have been repeated for them.

This description of the parser operation permits us to examine the parse of a string of lexical symbols corresponding to the example grammar. Let us use the string (a+b)*c$ where a, b, and c are specific instances of the identifier i in the grammar. A step by step narrative description of the parser action will be given.

1. Enter state R1, push R1 on stack, read (; a match is found with a transition to R6.

2. Enter R6, push R6 on stack, read a; a match is found (a is an identifier i) with a transition to A6.

3. Apply production 6 by popping 0 entries off stack. Use top state vector T6. Top state on stack is now R6 so make a transition to A4.

4. Apply production 4 by popping 0 entries off stack. Use top state vector T4. Top state in stack is now R6 so make a transition to L3.

5. Enter lookahead mode in state L3. Next symbol is + which belongs to default string, so make transition to A2. Leave lookahead mode.

6. Apply production 2 by popping 0 entries off stack. Use top state vector T2. Top state on stack is now R6 so make transition to R10.

7. Enter R10, push R10 on stack, read +; a match is found with a transition to R8.

8. Enter R8, push R8 on stack, read b; a match is found with a transition to A6.

9. Apply production 6 by popping 0 entries off stack. Use top state vector T6. Top state on stack is now R8 so make a transition to A4.

10. Apply production 4 by popping 0 entries off stack. Use top state vector T4. Top state on stack is now R8 so make a transition to L11.

11. Enter lookahead mode in state L11. Next symbol is) which belongs to default string, so make a transition to A3. Leave lookahead mode.

12. Apply production 3 by popping 2 entries off stack. Use top state vector T2. Top state on stack is now R6 so make a transition to R10.

13. Enter R10, push R10 on stack, read); a match is found with a transition to A7.

14. Apply production 7 by popping 2 entries off stack. Use top state vector T6. Top state on stack is now R1 so make a transition to A4.

15. Apply production 4 by popping 0 entries off stack. Use top state vector T4. Top state on stack is now R1 so make a transition to L3.

16. Enter lookahead mode in state L3. Next symbol is * which specifies a transition to R3. Leave lookahead mode.

17. Enter R3, push R3 on stack, read *; a match is found with a transition to R9.

18. Enter R9, push R9 on stack, read c; a match is found with a transition to A6.

19. Apply production 6 by popping 0 entries off stack. Use top state vector T6. Top state on stack is now R9 so make a transition to A5.

20. Apply production 5 by popping 2 entries off stack. Use top state vector T4. Top state on stack is now R1 so make a transition to L3.

21. Enter lookahead mode in state L3. Next symbol is $ which belongs to default string, so make a transition to A2. Leave lookahead mode.

22. Apply production 2 by popping 0 entries off stack. Use top state vector T2. Top state on stack is now R1 so make a transition to R2.

23. Enter R2, push R2 on stack, read $; a match is found with a transition to A1.

24. Apply production 1 by popping 1 entry off stack. Use top state vector T1. Top state on stack is now R1 so make a transition to state 0.

25. State 0 is the final state of the parser indicating a successful completion of the parse.

The parse tree which corresponds to parsing of the above example is shown in Figure A2.

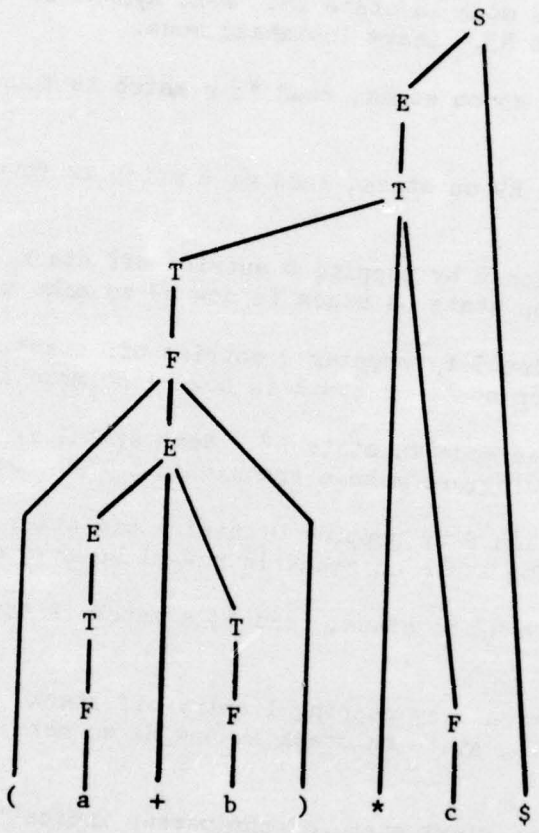


Figure A2. Parse Tree for Example Grammar

5. INCLUSION OF EMBEDDED ACTIONS IN PRODUCTIONS

The previous discussion has been concerned with the construction of LR(k) parsers in which no arbitrary actions have been permitted inside of the productions. Such a parser permits coupling with semantic routines and syntax directed translations after a production has been completed but prior to popping the stack. We now extend the parser to include arbitrary actions at any point during the formation of a production, including actions which occur before the first term of the production is processed.

There are two important applications for such arbitrary actions. First, it permits the user to specify error recovery information at any point during the parse. Second, the user has the capability of calling semantic routines at any point in the parse, not just at the end of a production.

We will discuss the extension of the parser in terms of action commands which may be embedded in the grammar. Action commands, represented by action symbols, will be handled in a manner similar to terminal symbols with some important differences. They will generate action states as opposed to read states. They do not enter into the lookahead process. On the other hand, they tend to create additional inadequate states in the CFSM, and lookahead strings considerably longer than the k symbols dictated by the underlying LR(k) grammar can arise in certain situations.

The action commands provide a means of interacting arbitrarily with the parsing operation. As such it is irrelevant whether they are used for semantic routine coupling purposes or for error recovery; the principal concern is a technique for coupling with the grammar of interest.

We start with a numbered set of productions, which may include action commands, with the production defining the grammar goal symbol occurring first. We now proceed to build a CFSM in the same manner as before by developing basic states and expanding these basic states whenever a non-terminal occurs. The action symbols, terminal symbols, and the # symbol do not lead to additional items in the state expansion.

We then classify the CFSM states as read states, action states, apply states, and inadequate states. The definitions for read and apply states are the same as before. An action state contains only a single action symbol. All other states are inadequate states. Thus we see that an inadequate state can represent any combination of terminal symbols, action symbols, and # symbols.

We now construct a DPDA in a manner similar to that used before. The terminal read transitions of CFSM read and inadequate states generate DPDA read states. Each # symbol in the CFSM generates an apply state in the DPDA. Each action symbol in the CFSM action and inadequate states generates a DPDA action state. The CFSM inadequate states generate DPDA lookahead states.

The top state vector now contains the initial read or action state identities for each production. This extension to actions states is essential since a production may specify an action command before the first grammar term. In order to keep the stack from containing superfluous information during parsing only the read states and any action state which is the initial term in a production are pushed onto the stack. Thus, when the production is applied, the usual n - 1 grammar symbols are popped if the production begins with a grammar symbol; one additional symbol is popped if the production begins with an action symbol.

The lookahead expansion differs considerably from that used when action symbols are not present. Whereas before lookahead was necessary only to decide whether a production should be applied or another terminal symbol should be read, (i.e., lookahead was associated only with the end of a production), now we must decide whether to perform an action, apply a production, or read a terminal symbol (i.e., lookahead may be associated with any point in a production).

The situation can be illustrated by the following example.

Suppose that we have the following two productions in the grammar:

A = w X v y
A = w Y v z

where A is a nonterminal, X and Y specify two different actions, and w, v, y, and z are strings of terminal and nonterminal symbols. Upon encountering the symbols X or Y we have no knowledge as to which production we are in, and thus have no basis for choosing X or Y to be executed. Thus, we must do a lookahead noting whether we ultimately encounter the string y or z in order to make the decision. Since the string of symbols v may be arbitrarily long we note that the amount of lookahead required may be quite large compared to that demanded by the intrinsic grammar.

In performing the lookahead expansion we separate the lookahead state into its basic substates; these may be read states, action states, and apply states. We then expand each basic substate until a unique symbol string has been constructed for each substate, thus resolving the lookahead problem. In this expansion any action state encountered automatically transitions to its destination state.

Expansion of an apply state into all of its potential top state vector destinations can lead to serious problems however. The problem can be illustrated with the following grammar:

B = a X C d
B = a C f
C = g

where A, B, and C are nonterminals, X is an action, and a, d, f, and g are terminals. We find upon doing the reduction for C that if we use all top state destinations that we have no capability of resolving the lookahead ambiguity, i.e., the choice of whether to read g or do action X cannot be resolved by a finite lookahead. But clearly such an impasse does not really exist. What is needed is to know which production B

was being used in the lookahead expansion so that only the top state associated with that production is used when the reduction to C is completed. This can be retained for each component of the original lookahead state by using a pushdown stack which stores that information as each read state is encountered. As a reduction is made, the appropriate stack is popped and a top state transition is made as though normal parsing were occurring. But notice that the stack may be exhausted before the lookahead is resolved. This means that the local context of the lookahead has been exhausted and that we have moved out into additional productions which could have called the production from which the lookahead situation was initiated. In this case we must now use all top states since we have no way of knowing which top state is the correct one. This is equivalent to the situation in resolving lookaheads for grammars which do not contain action specification.

6. ERROR RECOVERY

The error recovery mechanism depends upon the specification within the grammar of error recovery initiators and error recovery points. The initiators and recovery points always occur in pairs with the initiator preceding the recovery point in the grammar. Also, the members of a pair must occur in the same production. Pairs may be nested to any depth, providing the capability for hierarchical error recovery.

The error recovery initiator specifies a terminal symbol which the parser will search for in the input string in the event of an error. A recovery will be attempted at the point where the recovery symbol is located in the input string. However, it is also necessary for the parser to locate itself both in the parsing tables and in the DPDA stack so that it can resume the parse at the designated point in the input string as though nothing had gone wrong. The information necessary to adjust the parser is specified by the error recovery point in the production.

An error recovery stack is maintained in addition to the normal DPDA stack used by the parser. Under normal error free parsing, any error recovery initator action pushes its associated error recovery symbol onto the error recovery stack. When the corresponding error recovery point action is encountered, the appropriate error recovery symbol is popped from the stack (it will always be on the top of the stack in error free operation because of the nested pairing requirement for initiators and recovery points). Thus the recovery stack at any point in the parse represents a hierarchical error recovery system, with the top of the stack representing the most local error recovery capability and the bottom of the stack the most global capability.

When an error occurs the parser reports the error and immediately suspends parsing action. The lexical analyzer is then repetitively called using a standard scanner until a lexical symbol is delivered which matches an error recovery symbol in the error recovery stack. At this point error recovery is attempted. The DPDA stack and the parser table pointers are adjusted and parsing is resumed after popping all error recovery entries down to and including the matching entry in the recovery stack. In the event that recovery was incorrect another error will soon be reported and the recovery will be attempted again.

Since the DPDA stack position is known at the time the error recovery initiator action occurs, it is trivial to calculate the correct stack position at the corresponding error recovery point since both initiator and recovery point actions are in the same production with a fixed number of grammar symbols separating them. The location in the parsing tables for the error recovery point is also well known since it corresponds to a particular action state which is coupled to the initiator action state.

APPENDIX B

THE LR(k) PARSER GENERATOR

The LR(k) parser consists of a set of subroutines which are programmed by parsing tables to permit recognition of sentences in a particular language. The generation of the parsing tables is quite complex and tedious; thus it is desirable to have the parser table generation process automated. The LR(k) Parser Generator performs the function of automatically generating the LR(k) tables. The tables also include information concerning error recovery, semantic routine interfaces, and multiple syntax directed translations.

1. INPUT SPECIFICATION

Input to the parser generator is in the form of card images. The input deck is divided into two sections. The first section describes the lexical symbols, their lexical codes, and the assignment of lexical symbols to scanners. The second section describes the productions of the grammar using a BNF notation, along with error recovery mechanisms, semantic routine interfaces, and syntax directed translations. The input is terminated with an END card. Comment cards may be inserted at any place in the input deck. The input deck grammar is given in Appendix J.

a. Grammar Symbols

The grammar is specified in terms of lexical symbols (or terminal symbols) and nonterminal symbols. The user may compose alphameric identifiers to represent these symbols for purposes of describing the grammar. Each identifier may be of arbitrary length, must begin with a letter, and contain only letters and digits.

The user may also define any arbitrary string of symbols (including blanks) to represent a lexical symbol. The arbitrary symbol string is delimited by the symbol $ at the beginning and end of the character string. If the character $ is required in the character string, it is prefixed with another $ character which acts as an escape symbol.

The delimiting $ characters are merely used as endmarkers in identifying the desired character string and are not retained when the character string is internally stored. Note that $ABCDEF$ is identical to ABCDEF because of this fact.

Sequences of $ delimited character strings must be separated by a blank to prevent the trailing $ of one from becoming an escape character for the leading $ of the next string. For example, the sequence $+$ $*$ must not be written as $+$$*$; the last form represents the character string +$*.

b. Lexical Interface Description

The parser obtains lexical symbols from the lexical analyzer. The lexical analyzer may contain from 1 to 15 scanners, and a particular symbol may be obtainable from more than one scanner. Since the lexical analyzer and parser are separate software entities with tables which are independently generated, it is necessary for the Parser Generator to know from which scanners a particular lexical symbol may be obtained, and what lexical code is assigned to the lexical symbol by the lexical analyzer. In this way the Parser Generator is able to designate a particular scanner to be used at a given point in the parse, with assurance that all the possible legal symbols which can occur at that point can be delivered by the designated scanner. By knowing the lexical code assignment for the lexical symbols the parser is able to differentiate between the various choices and continue to the next state in the parsing algorithm.

The lexical interface description is designated on a scanner by scanner basis, using a sequence of scanner blocks, one for each scanner of interest. Each scanner block is introduced by a card with the format:

/LEXICAL n

where n = the scanner number. This is then followed by one or more cards which designate the lexical symbols which are obtainable from the designated scanner. Each lexical symbol card has the format

lexsymbol lexcode

where lexsymbol is the identifier chosen to represent the lexical symbol, and lexcode is the numerical lexical code assigned to that symbol.

c. Grammar Specification

The grammar specification lists all of the productions using BNF notation written in terms of the identifiers chosen to represent the nonterminals and lexical symbols. The grammar specification block is introduced by the card

/BNF

This card is immediately followed by the grammar productions, where the first production defines the grammar goal symbol. Each production is written in the form

nonterm = productiondef

where nonterm is a nonterminal, and productiondef is a sequence of nonterminal and lexical symbols which define the production. The nonterminal on the left hand side is called the production goal symbol. There may be as many productions with the same production goal symbol as desired; this is true even when the production goal symbol is also the grammar goal symbol. Productions may also be left recursive (the production goal symbol appears as the first symbol on the right hand side) or right recursive (the production goal symbol appears as the last symbol on the right hand side). Right recursive productions will be flagged with a warning message by the Parser Generator since such productions can lead to stack overflow during actual parsing.

An example of grammar specification for simple mathematical expressions is:

```
S = E
E = T
E = E $+$ T
T = F
T = T $*$ F
F = I
F = $($ E $)$
```

where S is the grammar goal symbol; S, E, T, and F are nonterminals; and $+$, $*$, I, $($, and $)$ are lexical symbols.

d. Error Recovery Specification

The LR(k) parser is completely deterministic and will announce syntax errors immediately. However, the recovery from a syntax error can be quite difficult. Although it is possible to devise automatic methods for recovering from errors (which may or may not result in recovery, depending upon the subtlety of the error), it was decided that the error recovery information should be incorporated by the user directly into the grammar description itself. This technique ensures eventual error recovery and usually recovers quickly with high reliability whenever the offending section of text has been passed over.

The error recovery mechanism consists of two special symbols, one to specify an error recovery symbol, the other to specify the recovery point in the grammar. The error recovery symbol designator specifies the lexical symbol which will be sought in the input text should an error occur. The recovery point symbol specifies the point in the grammar at which parsing will attempt to resume following the discovery of an error.

The error recovery problem involves the simultaneous handling of two problems. First, since the input text contains errors, it is necessary to skip over a portion of the text, starting at the point where the error was discovered, to a point where valid text can again be recognized. At the same time, the parser must adjust its stack contents and table pointers so that it is able to resume parsing as though nothing had gone wrong.

The error recovery symbol designator specifies the symbol which must be located in the input text should an error be encountered. This searching is done using scanner 1, so that it is imperative that scanner 1 be capable of delivering the lexical symbol which is being specified for error recovery. The recovery point designator indicates the point in the grammar at which error recovery will have occurred, corresponding to the successful acquisition of the recovery symbol in the text, and thus permits the parser to adjust its stack and table pointers correspondingly.

From the above description it can be seen that the two special symbols associated with error recovery occur in pairs, like left and right parentheses. Both members of a pair must occur in the same production. Pairs may be nested in the same way that parentheses may be nested. Thus it is possible to define a hierarchy of active error recovery specifications. The error recovery mechanism attempts to recover using the innermost error recovery pair; failing that, it will use the next innermost pair, and so on. Thus it can be seen that if the error recovery specification is well chosen that recovery from even badly garbled text can be effected.

The error recovery symbol is always a lexical symbol. It is specified at some point in the production and is enclosed in parentheses. The error recovery point is specified by the special symbol * at the appropriate point in the production. An example of a production with nested error recovery specification is

A = B (LP) C D (COMMA) E * F * G

Following recognition of symbol B the parser would set the lexical symbol LP into the error recovery symbol list. Following the recognition of symbol D the parser would set the lexical symbol COMMA into the error recovery list. If an error is encountered in trying to recognize C, D, or F, error recovery will be attempted by searching the input text for lexical symbol LP and resuming parsing with symbol G.

If an error is encountered in attempting to recognize E, a search will be made for lexical symbol COMMA; if found, parsing will continue with symbol F; however, if COMMA is not found but LP is found, then parsing will continue with symbol G.

Note that the grammar is not changed by the introduction of the error recovery information, i.e., in the above example the grammar still remains

A = B C D E F G

e. Semantic Routine Specification

At any point in the parsing of a production it may be desirable to interrupt the parse and call a semantic routine for doing some side processing. Normally, this is done at the completion of a production, just before the parser reduces the production to its goal symbol. The semantic routines are assigned integer numbers and a semantic routine can be called upon the completion of a production by using the following specification

```
A(56) = B C D
```

where semantic routine 56 is called.

It is also possible to call a semantic routine before the end of a production is reached, or even before the first symbol of the production is processed. In this case the semantic routine number is specified at the point in the production where the routine is to be called, with the number enclosed in parentheses as before.

```
A = B (45) C (3) D
B = (24) F G
```

When a semantic routine is called before the production end is reached, an extended lookahead situation can result. This happens because the parser may be working on several productions simultaneously and must look ahead to see which of the various productions it really is processing before it can know whether to call the semantic routine. Thus, for example, even though the underlying grammar may be LR(1) the parser may have to contend with lookahead equivalent to an LR(5) grammar, because of additional lookahead imposed by either internal semantic routine coupling or error recovery specification.

f. Syntax Directed Translation Specification

One or more syntax directed translations (SDTs) may be specified along with any production.

In syntax directed translation each production is represented by the ordered set (U, x, y) where U is the production goal symbol, x is a string of terminal and nonterminal symbols representing the definition of U, and y is a string of terminal and nonterminal symbols representing the translation of x. Since U is equivalent to x, U is also equivalent to the translation y of x. In BNF notation the ordered set is written in the form

U = x, y

In writing the string y we require that all nonterminals in y must also appear in x. The terminal symbols in y may be arbitrary. Furthermore, we associate a particular occurrence of a nonterminal in y with a particular occurrence of that nonterminal in x. If the nonterminals appear in the same order in x and y, this association is obvious since it is one-to-one. For example:

MTERM = MTERM $*$ MFACTOR, $*$ MTERM MFACTOR

Here we see that the first occurrence of MTERM in x corresponds to the first (and only) occurrence of MTERM in y, the first occurrence of MFACTOR in x corresponds to the first occurrence of MFACTOR in y, etc. Thus, the SDT consists of a mapping of nonterminals in one string to those in another string. There is no mapping permitted for the terminal symbols.

It is possible to specify mappings which are not one-to-one between x and y. This is done by assuming an implicit sequential numbering for each kind of nonterminal in x; y is then written with each nonterminal suffixed with the number of the corresponding x nonterminal enclosed in parentheses. For example:

A = B F B G C B C, H C(2) B(3) B(1) H B(3)

where A, B, and C are nonterminals, and F, G, and H are terminals.

Note that particular nonterminals may be reused or omitted with this scheme. Thus, the first occurrence of C in y corresponds to the second occurrence of C in x, the first occurrence of B in y corresponds to the third occurrence of B in x, etc. The terminals F and G in x have no relation to the terminal H in y.

The syntax directed translation can be generalized to permit several syntax directed translations to proceed in parallel with the parsing. Furthermore, each SDT is permitted to access information belonging to another SDT.

In multiple SDT each production is represented by an ordered set (U, x, T1, T2, ..., Tn) where U is the production goal symbol, x is a string of terminal and nonterminal symbols representing the definition of U, and T1, T2, ..., Tn are translations of x. The ith translation is represented by the ordered pair (i, y_i) where y is a string of terminal and nonterminal symbols.

In BNF notation a multiple SDT is written in the form

$$U = x,\ 1 = y_1,\ 2 = y_2,\ \ldots\ldots,\ n = y_n$$

In writing the strings y we require that all nonterminals in y must appear in x. Again, the terminal symbols in y may be arbitrary. We associate a particular occurrence of a nonterminal in y with a particular occurrence of that nonterminal in x; in addition, we associate the semantic value of a particular nonterminal in y with a particular semantic value of the nonterminal in x.

Note that each nonterminal in x has n semantic values associated with it, one for each translation involved in generating that nonterminal. Thus, the string y for each translation T may designate both the particular nonterminal occurrence in x and the particular semantic value attached to that nonterminal for purposes of constructing a particular translation. When the reduction to U is made, the n translations will correspond to the n semantic values to be attached to U, and thus the association is maintained.

We may append an ordered pair (j,i) to each nonterminal in the translation strings y, where j designates the nonterminal occurrence in x and i designates the semantic value of that nonterminal to be used in constructing y. If the nonterminals are referenced in the same order that they appear in x, the j index may be omitted by writing (,i); if the ith semantic value is used in the ith translation, the i index may be omitted by writing (j); and, finally, if both i and j can be omitted no ordered pair is required for designation.

An example of a multiple SDT is seen in the following example, where the production rule represents reduction of the product of a math expression term and factor to a term, the translation T1 represents the Polish prefix notation corresponding to the product, and translation T2 represents the Polish prefix notation for the total differential of the product:

```
MTERM = MTERM $*$ MFACTOR, 1 = $*$ MTERM MFACTOR, 2 = $+**$ MTERM (1,1)
   MFACTOR(1,2) MFACTOR(1,1) MTERM(1,2)
```

It is possible to include dynamically generated strings as part of SDT specifications. Such strings are created by semantic routine actions and then subsequently incorporated into a SDT operation. The dynamic string is specified in the SDT by a term of the form D.n where n is an integer. For example:

```
NETSTATE = RUNSOLN EOLC, $STATE$ D.1 EOLC
```

g. Comments

Comments may be inserted as desired in the input deck. Each comment must begin with the characters /COMMENT followed by a space. For example,

```
/COMMENT THIS IS A COMMENT CARD
```

h. Deck Structure

The parser generator input deck begins with the lexical interface specification, followed by the grammar specification, and concludes with the card

```
/END
```

i. Example Grammar Specification

The following example shows a complete Parser Generator input deck and includes error recovery, semantic routine interfaces, and multiple syntax directed translations. The underlying grammar is for simple mathematical expressions. The first translation is the Polish prefix

notation for the math expression, the second translation is the Polish prefix notation for the total differential of the expression, assuming that all lexical symbols I represent variables.

```
/COMMENT MATH EXPRESSION GRAMMAR EXAMPLE
/COMMENT SDT1 = POLISH EXPRESSION, SDT2 = POLISH DIFFERENTIAL EXPRESSION
/LEXICAL 1
$+$ 4
$*$ 5
$($ 6
$)$ 7
I 2
EOLC  10
/COMMENT EOLC IS END OF TEXT MARKER
/BNF
S(2)=(EOLC) E EOLC *, E, E
E = T, T, T
E = E $+$ (3) T, $+$ E T, $+$ E T
T = F, F, F
T = T $*$ (3) F, $*$ T F, $+$ $*$ $*$ T(1,1) F(1,2) F(1,1) T(1,2)
F(4) = I, I, I
F = (5) $($ ($)$) E $)$ *, E, E
/END
```

2. OPERATING DETAILS

The LR(k) Parser Generator is a CDC 6000 computer program which normally operates in 50000 octal words of core. Since the program utilizes dynamic storage allocation techniques larger amounts of core are required for large grammars.

Input to the program is through the system file INPUT in the form of card images. Listed output appears on the system file OUTPUT. The parsing tables appear as card images on file TAPE7.

3. LR(k) PARSER SUBROUTINES

The subroutines which comprise the LR(k) Parser are contained within the LR(k) Parser Generator software. The subroutines are LRKPRS, NXTSYM, BDLRK, RDCELL, APCEL1, APCEL2, LACELL, ERROR, and XSHIFT. The parsing tables are contained in a block data subroutine BDPARS which is automatically generated by the LR(k) Parser Generator.

The user must supply subroutine SEMANT if semantic processing is specified. SEMANT is called whenever a semantic routine is invoked by the parser. The semantic routine number is passed as an integer formal parameter to SEMANT through the linkage CALL SEMANT(ICODE).

The LR(k) Parser is invoked by calling subroutine LRKPRS. Control is not returned until completion of the parse.

APPENDIX C

LEXICAL DIGRAM ANALYSIS

The lexical analyzer is required to recognize lexical constructs which are formed by groups of characters in the input text stream. The rules for grouping may be very complex and change from scanner to scanner within the lexical analyzer. The lexical analyzer cannot be specified without a knowledge of the possible adjacent occurrences for lexical constructs which are imposed by the grammar of the language. Once the adjacent lexical constructs are known the termination rules for each construct can be specified.

An example of a lexical adjacency problem may be found in FORTRAN. Consider the character sequence DØ10I=J.... where the dots represent input characters not yet seen by the lexical analyzer. At this point we do not know whether a DØ statement or a replacement statement is being read. Yet the lexical analyzer has been asked by the parser to provide a lexical symbol for the first lexical construct, which may be either DØ or DØ10I. Another example is provided by the sequence IF(2.E..... where it is not obvious whether the E is the start of an exponent or of the .EQ. Boolean relational operator.

Lexical digram analysis permits the identification of all permissible pairs of adjacent lexical constructs in the grammar for the purposes of devising termination rules for the first member of the pair in the particular scanner which recognizes that member. However, since the digrams arise from the grammar, digram analysis must be performed as part of the grammar analysis by the Parser Generator.

In the above examples the digram analysis would reveal, among other things, that the keyword DØ may be followed by an integer label, the combination being indistinguishable from a single FORTRAN identifier; thus additional lexical lookahead is required to resolve the ambiguity. Similarly, digram analysis would indicate the juxtaposition of a number with a Boolean relational operator, from which the problem of determining the proper termination of the lexical construct for a number becomes evident.

Lexical digrams are formed by constructing all the possible adjacencies of terminal symbols in the grammar. However, terminal symbols may become adjacent by any of four mechanisms as viewed from the grammar specification. We can formally calculate the digrams by the use of incidence matrices, adjacency matrices, and transitive closures of these matrices.

Let us assume a grammar G given by the 4-tuple G(N,T,P,S) where N = the set of nonterminal symbols, T = the set of terminal symbols, P = the set of productions, and S = the grammar goal symbol. We wish to find the set of digrams of T which can occur in the language generated by the grammar G.

We define a matrix F whose rows are labeled by N and whose columns are labeled by N and T. This matrix can be partitioned into F_{NN} and F_{NT}. An element f_{AB} of the matrix f = (1, 0) if there (does, does not) exist a production in P of the form A = B Q where Q may be any arbitrary string of terminal and nonterminal symbols including the empty string. Thus the matrix F defines the first terminal or nonterminal symbol(s) which begin a production(s) which defines the set of nonterminals.

We next define a matrix X whose rows and columns are labeled by N and T. This matrix can be partitioned into X_{NN}, X_{NT}, X_{TN}, and X_{TT}. An element x_{AB} of the matrix X = (1, 0) if there (does, does not) exist a production in P of the form U = Q A B R where Q and R may be any arbitrary string of terminal and nonterminal symbols including the empty string. Thus the matrix X defines the sets of adjacent symbols in the productions P.

We also define a matrix L whose rows are labeled by N and whose columns are labeled by N and T. This matrix can be partitioned into L_{NN} and L_{NT}. An element l_{AB} of the matrix L = (1, 0) if there (does, does not) exist a production in P of the form A = Q B where Q may be any arbitrary string of terminal and nonterminal symbols including the empty string. Thus L describes the last symbols N and T which may occur in the productions P which define N.

The terminal digrams can arise from four sources, representing the four partitions of X. Obviously the partition X_{TT} contributes terminal digrams due to their explicit occurrence in the productions.

The matrix X_{TN} represents the occurrence of a terminal symbol followed by a nonterminal symbol. Since all nonterminal symbols must be expandable to a string of terminal symbols, we see that there will also be a digram contribution from the X_{TN} matrix. All that is required is to find the first terminal symbols which occur as the result of the nonterminals. This information is given by the transitive closure of the matrix F_{NT}; the transitive closure of this matrix is denoted by F^{+}_{NT}. Thus, the product $X_{TN}F^{+}_{NT}$ contributes the terminal digrams due to the X_{TN} matrix partition.

The X_{NT} matrix describes nonterminals followed by terminals. Again, the nonterminals lead to strings of terminal symbols. In this case we are interested in the last terminal symbols in the strings generated by the nonterminals; this is given by the transitive closure L^{+}_{NT}. However, we need the transpose of this matrix, or L^{+t}_{NT} which gives the nonterminals which correspond to the last terminal symbols of the string. Then the product $L^{+t}_{NT}X_{NT}$ gives the terminal digrams arising from the matrix X_{NT}.

Finally, the matrix X_{NN} can make terminal digram contributions. Here we have the adjacency of two nonterminals in a production, both of which lead to terminal strings. We are interested in the last terminal of the string produced by the row index of X_{NN} and the first terminal of the string produced by the column index of X_{NN}. Thus, the digram contribution of X_{NN} is given by $L^{+t}_{NT}X_{NN}F^{+}_{NT}$.

The complete digram incidence matrix for terminals, D_{TT}, is then given by the logical union of the four terminal digram sources:

$$D_{TT} = X_{TT} + X_{TN}F^{+}_{NT} + L^{+t}_{NT} (X_{NN}F^{+}_{NT} + X_{NT})$$

where the element d_{AB} of D_{TT} = (1, 0) if the ordered terminal symbol pair A B (does, does not) exist in the language described by the grammar G.

APPENDIX D

THE LEXICAL ANALYZER GENERATOR

The lexical analyzer consists of a set of subroutines which are programmed by a set of tables to permit recognition of specific lexical constructs associated with a particular language. The manual generation of the lexical analyzer tables is not easily accomplished; therefore, it is useful to generate these tables automatically using the Lexical Analyzer Generator.

1. INPUT SPECIFICATION

Input to the Lexical Analyzer Generator is in the form of card images and consists of three sections. These sections specify the lexical analyzer constants, the lexical interface, and the scanner definitions. Appendix L describes the grammar for the Lexical Analyzer Generator input.

a. Constant Specification

This section of the input assigns values to constants which are used by the lexical analyzer and describes the character codes which are to be used as input to the executing lexical analyzer from the character generator subprogram.

The maximum scanner number which will be used by the lexical analyzer is specified by:

/MAXSCAN n

where n is the maximum scanner number. Scanner numbers may be assigned from 1 to 15.

The lexical tables may include semantic workspace for each identifier in the ISTAB array. The user may specify the size of the workspace area with:

/WORKSPACE n

where n is the number of words to be allocated for workspace for each identifier. The workspace area begins with the second word of each identifier cell in ISTAB. A workspace length of zero may be assigned. Appendix Q discusses the ISTAB array in more detail.

Keywords and identifiers are accessed by the lexical analyzer using a hash table lookup. The length of the hash table may be specified by:

/HASHTABLE n

where n is the hash table length. The hash table length must be a prime number in order for the hashing algorithm to work properly. The hash table length should exceed the maximum number of keywords and identifiers to be stored by approximately 10-15% in order to maintain reasonably efficient lookup.

The keywords and identifiers are stored in the ISTAB array in a set of cells, one cell for each unique keyword or identifier. Cell length is variable and is given by the formula:

N = W + S + 1

where N is the cell size in words, W is the workspace size as specified by the user, and S is the length of the keyword or identifier string in words. Keywords and identifiers may be of arbitrary length. Characters are stored in the string using eight characters per word, and a special terminating character is always automatically added to the end of the string. Thus, a string of seven characters requires one word, whereas a string of eight characters requires two words (the second word contains only the special terminating character). The user may specify the total length of the ISTAB array which contains all of the keyword and identifier cells by:

/IDTABLE n

where n is the length of the ISTAB array in words.

The lexical analyzer reads characters delivered by the character generator subprogram as input. The characters read by the lexical analyzer are not restricted to the standard keypunch characters but may be any arbitrary character set. The characters are represented internally by character codes, using the positive nonzero integers. The maximum character code value is specified by the user by:

/CHARCODE n

where n is the maximum character code value.

Immediately following the specification of the CHARCODE constant the user must specify the character codes which are to be used. The character codes are specified by listing the character name followed by the integer code value, using one character code specification per card. Character names may be represented by alphameric identifiers of arbitrary length, provided that each name begins with a letter. A blank is used to separate the character name from the code value. For example:

```
A  1
B  2
D1  24
D2  25
BLANK  45
EOF  50
```

Comments may be included at any point within the constant section except between /CHARCODE and the end of the character code list. A comment is represented by the keyword /COMMENT followed by a blank and an arbitrary comment text. For example:

```
/COMMENT  THIS IS A COMMENT
```

Comments may not be continued from one card to the next, although consecutive comment cards may be used.

b. Interface Specification

The output of the lexical analyzer is a series of lexical codes, one for each call to the analyzer, representing lexical constructs as they are recognized from a normal left to right scan of the input sentence. The lexical codes are used as input to the parser which is responsible for recognizing correct grammatical constructs of the language being read.

The lexical interface section specifies the names of the lexical symbols and their codes. All specifications in this section are global specifications, i.e., they apply to all scanners which are capable of recognizing the lexical construct which do not have local specifications which override the global specification.

The lexical interface specification is introduced with the card:

```
/LEXCODE
```

This is followed by one or more lexical symbol definitions, one definition per card. Each lexical symbol is defined by listing the symbol name followed by its lexical code, using a blank to separate the name from the code. Lexical symbol names are identifiers of arbitrary length and are formed using the letters A through Z, the digits 0 through 9, and the characters + - ( ) = . / $ * provided that first character of the identifier is not a digit or a / $ * character. The lexical codes are positive nonzero integers. The same names and codes may be used for lexical symbols as were used for character code definition without confusion. An example of lexical symbol definition is:

```
EXPONENT  45
SLASH  36
D2  109
.EQ.  53
```

The user may also specify optional semantic information for a lexical symbol. The semantic information is useful in complicated situations where several different keywords must have the same lexical code, yet the semantic routines must be able to distinguish certain properties associated with the actual keywords; these properties can be coded into a semantic bit field for the lexical symbol. The semantic information may be defined globally as described here or locally under an individual scanner, and it may be changed from scanner to scanner for a particular keyword.

The semantic information is microprogrammed, using a 29 bit field in the first word of the keyword or identifier cell in the ISTAB array. Microprogramming is accomplished through a cursor which can be shifted left in the 29 bit field; initially it starts at the right edge of the field.

For example, suppose we define keyword MAX with lexical code 35 and semantic value 13. This is done by:

```
MAX  35  13
```

which places the value 13 right adjusted in the semantic bit field. We can alternately specify insertion of integers into the semantic field and shifting of the cursor left in the field; each time the cursor is shifted it redefines the right hand edge of the semantic subfield for the next integer. For example:

```
MAX  35  13  L6  8  L5  2
```

This defines a semantic field containing three bytes. The rightmost byte is 6 bits wide and contains 13, the middle byte is 5 bits wide and contains 8, and the left byte fills the balance of the semantic field and contains 2.

The user may include keywords without any lexical or semantic codes in the lexical interface section by simply listing the keyword. Such a listing produces no entries in the lexical analyzer tables and this feature is available to permit keywords to be globally listed when their lexical codes are locally defined under individual scanners.

Comments may not appear in the lexical interface section.

c. Scanner Specification

The lexical analyzer is composed of one or more scanners, each of which has different properties and is capable of recognizing different sets of keywords and other lexical constructs. When the parser calls the lexical analyzer it specifies a particular scanner which is to be used by the lexical analyzer. This scanner must have the capability of recognizing all of the lexical constructs which can appear at that point in the parse. The scanners may be quite complex and it is possible for a scanner to invoke another scanner to assist in the recognition of lexical constructs.

Each scanner is separately specified, with a maximum limit of 15 scanners permitted. A scanner specification is introduced by:

```
/SCANNER  n
```

where n is the scanner number, an integer from 1 through 15.

A scanner specification may be composed of five subsections, which specify the character classes, keywords, keyword lengths, a finite state automaton, and an identifier code. These subsections may appear in any order.

Comment cards may be included in the scanner specification; however, they may appear only where a subsection card may appear.

(1) Character Class Specification

The scanner operates as a finite state machine with one character of lookahead. The scanner is composed of eight subscanners, any of which may be used to generate a symbol. The choice of subscanner is made on the basis of the first character used in generating the symbol. By means of a table lookup, indexed by the numerical code representing the first character, a choice of subscanner is made. The various subscanners have different properties, including the ability to transfer control to another subscanner, another scanner, or an external routine.

Let us define eight classes, one for each subscanner. We will categorize the initial character which is used to generate a symbol as belonging to one of the eight classes. Thus, when the initial character is read, it automatically initiates the action of one of the eight subscanners.

All characters which cannot be valid initial characters for generating a symbol are assigned to Class 1. When a Class 1 character is encountered as the initial character, a symbol code value = 0 is passed to the parser. This code value indicates an illegal symbol and the parser announces an error in the input sentence.

All characters which can be used as the initial character in forming keywords or identifiers are assigned to one of Classes 2, 3, and 4. The particular class which is chosen depends upon the requirements for recognizing keywords and/or identifiers.

The subscanner initiated by a Class 2 character attempts to construct the longest permissible keyword, using specifically designated keyword lengths. If the keyword is not identified, the subscanner then forms the longest possible identifier using the subscanner for Class 4. In building keywords, the Class 2 subscanner consults a second table which specifies whether a particular character can appear as a valid character within the main body of the keyword or identifier. If a non-keyword/identifier character is encountered, construction of the keyword/identifier is terminated; any keyword recognized up to that point is delivered to the parser, otherwise, an identifier is delivered. The Class 2 subscanner is useful for those situations in which keywords appear buried in the midst of other non-keyword constructs which use the same characters as the keywords. An example from FORTRAN is the string DATAIN in which DATA could be a keyword and IN an identifier for DATA statements, but DATAIN is an identifier for replacement statements.

The subscanner initiated by a Class 3 character is very similar to the Class 2 subscanner. It also attempts to recognize a keyword. However, the keyword is of unspecified length and is terminated only by encountering a non-keyword/identifier character. If upon termination the construct does not correspond to a keyword, the subscanner delivers it as an identifier as before.

The subscanner initiated by a Class 4 character is identical to the Class 3 subscanner except that it does not look for keywords at all; it always treats the construct as an identifier, even though it may also be a keyword.

The subscanner initiated by a Class 5 character is a finite state automaton (FSA) which may be programmed to meet a variety of needs. The FSA subscanner is an extremely powerful and flexible component of the lexical analyzer. It can handle many difficult lookahead situations encountered in symbol generation. The FSA is discussed later in this appendix.

Any character which is to be completely ignored by the lexical analyzer (such as the blank in FORTRAN) is categorized as a Class 6 character. The scanner immediately reads the succeeding character and proceeds with its normal action. Class 6 characters encountered by the Class 2, 3, and 4 subscanners are also ignored.

Any character which terminates an identifier or keyword but is otherwise ignored is a Class 7 character. An example is the blank in NET-2 which acts as a separator for keywords and identifiers. The scanner action is identical to that for a Class 6 character, except that in the Class 2, 3, and 4 subscanners the keyword or identifier formation is terminated. Symbol codes for Class 6 and 7 characters are never delivered to the parser.

The subscanner initiated by a Class 8 character calls an external subroutine, selected by a designated numerical code. The external subroutine is able to provide any sort of lexical analysis which is necessary. An example is the concatenation in FORTRAN of the various real number representations followed by the Boolean relational operators. It is difficult to know whether the dot is a period or a decimal point and whether the E is the introduction of the exponent or part of the .EQ. Boolean relational operator. The subroutine may set flags specifying a restart of the character string scan using an alternate subscanner or scanner.

In summary, the subscanners may be described by:

1 Illegal constructs
2 Fixed length keywords
3 Arbitrary length keywords
4 Identifiers
5 Finite State Automata processing
6 Ignore character, do not terminate lexical construct
7 Ignore character, terminate lexical construct
8 External lexical processing

The character class subsection is introduced by:

```
/CLASS
```

followed by one or more cards which assign characters to character classes (and thence to subscanners). Each assignment card lists the class number followed by one or more characters which are assigned to that class. For example:

```
3  A  B  C  D  SLASH  D6
5  P  Q  STAR  DOT
6  BLANK
8  EOF  COMMA
```

All characters which are permitted in keywords and identifiers following the initial character are also assigned to character class 9. Thus, a scanner which can recognize the alphameric identifiers permitted in FORTRAN (including the ignored blank character) is described as follows:

```
/CLASS
4  A  B  C  D  E  F  G  H  I  J  K  L  M  N  O  P  Q  R  S  T  U  V  W  X  Y  Z
6  BLANK
9  A  B  C  D  E  F  G  H  I  J  K  L  M  N  O  P  Q  R  S  T  U  V  W  X  Y  Z
9  D0  D1  D2  D3  D4  D5  D6  D7  D8  D9
```

(2) Keyword Specification

The keyword subsection lists all lexical constructs which are to be recognized as keywords by the scanner. If the keyword and its lexical code have been defined globally it is only necessary to reference the keyword here. However, if the keyword has not been globally defined or it is desired to specify a local definition of the keyword, then a complete keyword specification must be made.

The keyword subsection is introduced with the card:

```
/KEYWORDS
```

This is followed by keyword specification or reference cards, one for each keyword of interest. A keyword specification is made in exactly the same way as the global specification, including semantic information if desired. A keyword reference only lists the keyword itself without any lexical code or semantic information.

(3) Keyword Length Specification

If the scanner is required to recognize fixed length keywords it is necessary to specify the fixed lengths which correspond to those keywords. This is done with a single card which begins with /KWLENGTH followed by a set of integers in ascending order which specify the keyword lengths. Blanks are used as separators. For example, if the fixed length keywords are PROGRAM, FUNCTION, and SUBROUTINE, then the keyword length specification is:

```
/KWLENGTH  7  8  10
```

The fixed length keywords are useful in isolating keywords which are not terminated by a non-keyword character. In this way the keyword FUNCTION can be identified in the text string FUNCTIONRANDOM.

(4) Finite State Automaton Specification

The subscanner initiated by a Class 5 character is a finite state automaton (FSA) which may be programmed to meet a variety of needs. The FSA consists of a set of cells which are interconnected to form a state diagram. Each cell contains a symbol field, a match action, and a non-match action. The symbol field may represent either a specific character code or a class code. If the input character matches the symbol (in character code or class code, depending upon which type of match is indicated) the FSA performs the action specified by the match action. If the designated match does not occur, the FSA performs the action specified by the non-match action. If a match is found, the FSA reads a new character from the character generator; if there is no match, the previous character is retained for matching in subsequent FSA cells.

The FSA action may be one of four types. Type 0 is a transistion to another FSA state (including the state the FSA is currently in). Type 1 exits the scanner and lexical analyzer and returns a designated symbol code to the parser. Type 2 is a restart of the charcter string scan using a designated class for the first character; this overrides the automatic class assignment for the initial character and forces the scanner to restart in the designated class. Type 3 is a restart of the character string scan using a designated scanner; the subscanner of the new scanner is selected in the normal way, namely, by the class of the initial character using the class table of the new scanner.

The FSA subsection is introduced by the card:

/FSA

This is followed by one or more cards which define the FSA states.

Each state of the FSA is described by a card with the format:

$n, c, p, q

where n = an integer specifying the FSA state number, c = the character field, p = the match action field, and q = the non-match action field. The initial state number is 1.

The character field c has one of two forms. It specifies either the character code or class to which the current character is to be compared for a match. If the character code is to be compared then c contains the character name. If the character class is to be compared, then c contains the class number.

The match and non-match action fields may have any of four forms. If a transition to a new FSA state is desired, then the field contains $n where n is an integer specifying the desired FSA state number. If exit from the lexical analyzer is desired, then the field contains the name of the lexical symbol whose lexical code is to be delivered to the parser. If restart with a new subscanner is desired, then the field contains an integer specifying the new subscanner. If restart with a new scanner is desired, then the field contains *s where s is an integer specifying the new scanner number.

(5) Identifier Code Specification

The user must assign the lexical code to be delivered by the scanner when an identifier is recognized. This is done by a single card of form:

/IDENTIFIER n

where n is an integer specifying the lexical code for identifiers. If the identifier code specification is omitted the scanner will automatically deliver a code = 0 for identifiers, which is interpreted by the parser as a syntax error. Each scanner may use a different identifier code.

d. Deck Structure

The Lexical Analyzer Generator input deck begins with the lexical analyzer constants, followed by the lexical interface specification, followed by the scanner specifications, and concludes with the card:

/END

2. OPERATING DETAILS

The Lexical Analyzer Generator is a CDC 6000 computer program which operates in 50000 octal words of core. Since the program utilizes dynamic storage allocation techniques larger amounts of core may be required for very complex lexical analyzer specifications.

Input to the program is through the system file INPUT in the form of card images. Listed output appears on the system file OUTPUT. The lexical analyzer tables appear as card images on file TAPE7.

3. LEXICAL ANALYZER SUBROUTINES

The subroutines which comprise the lexical analyzer are contained within the Lexical Analyzer Generator software. The subroutines are LEXAN, LOOKUP, ADDCHR, and XSHIFT. The lexical analyzer tables are contained in a block data subroutine BDLEX which is automatically generated by the Lexical Analyzer Generator.

The user must supply a subroutine CHRGEN which supplies characters to the lexical analyzer from some external data structure. The character codes supplied must agree with the codes assigned internally in the lexical analyzer for the characters. The character generator is normally required to have backtrack capability as required by the Class 2, 5, and 8 subscanners.

The Class 8 subscanner requires the user to supply subroutine EXTSCN to accomplish specialized external scanning. EXTSCN is required to provide both the lexical and semantic code information to the lexical analyzer.

The lexical analyzer is automatically invoked by the LR(k) Parser software through the primary subroutine LEXAN.

APPENDIX E

CSL PARSER DESCRIPTION

This appendix lists the BNF input to the LR(k) Parser Generator which produces the LR(k) parsing tables for CSL. The input listing constitutes a complete description of the grammar for CSL, including identification of all terminal symbols and their lexical codes. The grammar is LR(2) and can be represented by 272 read states, 199 apply states, and 48 lookahead states.

A listing of the undefined terminal symbols and their corresponding character strings is given below:

Lexical Symbol	Character String
AND	.AND.
BOUNDCARD	BOUNDS
COMMENT	Any arbitrary comment text ending with EOLC character
ENDDECK	END DECK
ENDENTER	END ENTER
ENDRUN	END RUN
ENDSTATE	END STATE
EOLC	End of logical card symbol
FPNAME	Any legal function or procedure name
IAA	Any identifier containing only alphameric characters
IC	INITIAL CONDITIONS
ILA	Any identifier containing alphameric characters, starting with a letter
ILL	Any identifier containing only letters
INTEGER	An unsigned integer
KW1	SENS
KW2	N, NM, ND, NP, NR, or NI
KW4	I, V, P, IM, VM, PM, ID, VD, PD, IP, VP, PP, IR, VR, PR, II, VI, or PI
KW5	AM, BM, YM, ZM, AD, BD, YD, ZD, AP, BP, YP, ZP, AR, BR, YR, ZR, AI, BI, YI, or ZI

Lexical Symbol	Character String
KW5A	M, D, P, R, or I
KW6	Any element parameter name
KW7	Any global name
KW8	Any element prefix
KW9	COMPLEX, RADIANS, or DEGREES
KW10	PRINT
KW11	USE TEXT
KW12	A run control character string
KW13	A run control character string
KW14	A run control character string
KW15	A run control character string
KW16	A run control character string
KW17	A run control character string
KW18	A run control character string
KW19	PLOT
KW20	LINLOG, LOGLIN, LOGLOG, POLAR, or NYQUIST
KW21	LOG
KW22	FINAL
KW23	CURVE
KW24	ENTER NET, ENTER SCEPTRE, or ENTER CIRCUS
KW25	TABLE
KW26	2D
KW27	R
KW28	PROCEDURE
KW29	SUBNET
KW30	TEXT
KW32	S.
KW33	GAUSS, RECT, or TRI
KW34	OBJ
KW35	X
KW48	DATUM
KW49	PT, PV, IV, or D
KW50	E, F, or Q

Lexical Symbol	Character String
LANG	NET, SCEPTRE, or CIRCUS
MATHF	Any standard math function such as SIN, COS, etc.
MCCARD	RUN MONTECARLO
NOT	.NOT.
NUMBER	Any real number representation
OR	.OR.
REALNAME	Any legal real variable name
RELOP	.EQ., .NE., .LT., .LE., .GT., or .GE.
RUN	RUN
RUNOPT	RUN OPTIMIZATION
RUNWC	RUN WORST CASE
STATECARD	STATE
SUBNETEND	END SUBNET
SUBNETNAME	Any legal subnetwork name prefix
TEXTLIST	Any arbitrary text with END EOLC as last card image

The lexical analyzer description for the CSL grammar is given in Appendix F. The complete CSL grammar including CSL Translator directives is:

```
/COMMENT GRAMMAR FOR FULL CSL 7/25/75
/LEXICAL 1
KW22 92
$=$ 17
$TARGET$ 126
KW21 91
$,$ 63
KW27 97
KW33 103
$($ 15
KW20 90
KW26 96
LANG 127
NUMBER 53
KW23 93
$)$ 16
KW35 105
EOLC 10
```

```
/LEXICAL 2
$($ 15
ILA 52
$+$ 50
$-$ 67
$$$$ 134
EOLC 10
$,$ 63
$/$ 14
$*$ 64
$**$ 65
$=$ 17
$.$ 18
$)$ 16
AND 60
OR 59
NOT 62
RELOP 61
/LEXICAL 3
KW4 74
$*$ 64
SUBNETNAME 131
KW50 120
KW32 102
KW49 119
REALNAME 132
FPNAME 135
NUMBER 53
COMMENT 129
KW12 82
KW13 83
KW14 84
KW15 85
KW16 86
KW17 87
KW18 88
KW25 95
KW7 77
KW28 98
SUBNETEND 112
KW34 104
IC 111
$DEVICE$ 125
RUNWC 110
$END$ 69
$ENDMODEL$ 124
BOUNDCARD 113
KW8 78
KW19 89
$MODEL$ 123
```

```
RUNOPT 114
MCCARD 108
ENDDECK 107
ENDSTATE 116
KW24 94
KW11 81
KW30 100
ENDENTER 121
KW23 93
KW29 99
KW48 118
STATECARD 115
RUN 117
KW10 80
ENDRUN 109
/LEXICAL 4
KW23 93
$/$ 14
$,$ 63
EOLC 10
$+$ 50
$-$ 67
$($ 15
MATHF 122
KW1 71
KW32 102
KW49 119
KW7 77
KW5 75
KW4 74
KW22 92
KW50 120
KW2 72
REALNAME 132
SUBNETNAME 131
KW8 78
NOT 62
$$$$ 134
KW20 90
KW9 79
FPNAME 135
NUMBER 53
/LEXICAL 5
ILL 55
/LEXICAL 7
IAA 54
$/$ 14
KW8 78
SUBNETNAME 131
/LEXICAL8
KW6 76
/LEXICAL 9
```

```
TEXTLIST 130
/LEXICAL 12
$($ 15
KW5A 128
INTEGER 2
/BNF
S=TARGLANG DECK DECKEND
ABOUNDENT=$,$ KW23
ABOUNDENT=$,$KW33
ABOUNDENT=$,$ KW21
ABOUNDENT=$,$ KW23 $,$ KW21
ABOUNDENT=$,$ KW33 $,$ KW21
ABOUNDLIM=NUMBER$*$
ABOUNDLIM=ME
AELEMENT=ELEM $,$ NODELIST
ANODE=NODE
ANODE=$/$NODE
ANODEAC=$-$NODE
AOBJ=$,$ CURVEID $,$ NUMBER
AOBJ=$,$ CURVEID $,$ NUMBER $,$ KW35
AOP=$+$
AOP=$-$
AOUTLST=$$$$ IAA $$$$
APARAM=NUMBER $*$
APARAM=ME
ARG1=NUMBER
ARG1=OUTVAR
ARGLIST=ILA
ARGLIST=ARGLIST$,$ILA
ASWNUM=NUMBER
ASWNUM=NUMBER $*$
BELEMENT=$=$VALUES
BELEMENT=$($VALUELIST2$)$
BEXP=BTERM
BEXP=BEXP OR BTERM
BFACTOR=ME RELOP ME
BFACTOR=$($BEXP$)$
BFACTOR=NOT BFACTOR
BNODEAC=$/$NODE ANODEAC
BNODEAC=$/$ NODE
BOUND=BOUNDCARD EOLC BOUNDLIST $END$ EOLC
BOUNDENT=GRP1
BOUNDENT=INDVAR $=$ ABOUNDLIM $,$ ABOUNDLIM ABOUNDENT
BOUNDLIST=BOUNDENT EOLC
BOUNDLIST=BOUNDLIST BOUNDENT EOLC
BSWNUMLIST=$($ INTEGER KW21 $)$
BSWNUMLIST=$($ INTEGER $)$
BTERM=BFACTOR
BTERM=BTERM AND BFACTOR
CURVE=CURVEID EOLC NUMLIST
CURVEID=KW23
DECK=ENTRY
DECK=DECK ENTRY
```

```
DECKEND=$END$ EOLC
DECKEND=ENDDECK EOLC
DEPVAR=INDVAR
DEPVAR=IDRV
DEPVAR=FDRV
DEPVARLST=DEPVAR
DEPVARLST=DEPVARLST$,$DEPVAR
DNODE=KW48$,$NODE
DNODE=DNODE$,$NODE
DUMNODELIST=IAA
DUMNODELIST=DUMNODELIST$,$IAA
ELEM=KW8
ELEMENT=AELEMENT BELEMENT
ELEMENT=AELEMENT
ELNAME=ELEM
ELNAME=NEST$.$ELEM
ELVAL=ELNAME
ELVAL=ELNAME$,$KW6
ENTRY=GRP3
ENTRY=INTCOND
ENTRY=CURVE
ENTRY=KW24 EOLC TEXTLIST
ENTRY=$DEVICE$ $,$ IAA $,$ ILL $($ VALUELIST2 $)$ EOLC
ENTRY=$MODEL$ $,$ILL$,$NODELIST$=$ARGLIST EOLC MODBLK$ENDMODEL$EOLC
ENTRY=MCCARD EOLC MCLIST RUNEND
ENTRY=KW30 INTEGER EOLC TEXTLIST
ENTRY=RUN EOLC NSLIST RUNEND
ENTRY=RUNOPT EOLC OPTLIST RUNEND
ENTRY=RUNWC EOLC WCLIST RUNEND
ENTRY=GRP2
ENTRY=GRP1
FDRV=KW2 KW5A $($ NODE $)$
FDRV=KW4 KW5A $($ ELNAME $)$
FDRV=KW5 KW5A $($ NODEAC $)$
GRP1=KW11 INTEGER EOLC
GRP1=COMMENT
GRP2=SUBNETREF EOLC
GRP2=SUBNETCARD EOLC SUBNETLIST SUBNETEND EOLC
GRP2=FPNAME $($ ARGLIST $)$ $=$ ME EOLC
GRP2=TABLE
GRP2=ELEMENT EOLC
GRP2=KW28 $,$ FPNAME $($ ARGLIST $)$ EOLC TEXTLIST
GRP2=DNODE EOLC
GRP2=REALVAL $=$ ME EOLC
GRP3=BOUND
GRP3=RC EOLC
GRP4=INTCOND
GRP4=PARAM EOLC
INDVAR=ELVAL
INDVAR=KW7
INDVAR=REALVAL
INDVARLST=INDVAR
INDVARLST=INDVARLST$,$INDVAR
```

```
INTCOND=IC EOLC INTCONDLIST $END$ EOLC
INTCONDLIST=INTCONDVAR EOLC
INTCONDLIST=INTCONDLIST INTCONDVAR EOLC
INTCONDVAR=KW4 $($ KW8 $)$ $=$ ME
INTCONDVAR=KW32 $($ REALVAL $)$ $=$ ME
INTCONDVAR=GRP1
MCENT=CURVE
MCENT=GRP3
MCENT=GRP1
MCENT=GRP4
MCENT=MCOUT EOLC
MCLIST=MCENT
MCLIST=MCLIST MCENT
MCOUT=KW10$,$OUTLST
MCOUT=KW19$,$OUTLST
ME=AOP MTERM
ME=MTERM
ME=ME AOP MTERM
MFACTOR=MFACTOR1
MFACTOR=MFACTOR1 $**$ MFACTOR1
MFACTOR1=$($ ME $)$
MFACTOR1=MFUNC $($ VALUELIST1 $)$
MFACTOR1=ARG1
MFUNC=MATHF
MFUNC=FPNAME
MODBLK=MODENT
MODBLK=MODBLK MODENT
MODENT=GRP2
MODENT=GRP1
MODENT=INTCOND
MOP=$*$
MOP=$/$
MTERM=MFACTOR
MTERM=MTERM MOP MFACTOR
NEST=SUBNETNAME
NEST=NESTS.$SUBNETNAME
NODE=IAA
NODE=NESTS.$IAA
NODE=NEST $.$ KW8 $.$ IAA
NODE=KW8 $.$ IAA
NODEAC=NODE ANODEAC BNODEAC
NODEAC=NODE BNODEAC
NODEAC=NODE ANODEAC
NODEAC=NODE
NODELIST=ANODE
NODELIST=NODELISTS-$ANODE
NSENT=RC EOLC
NSENT=GRP1
NSENT=GRP4
NSENT=OUT EOLC
NSENT=$*$ SWVAR EOLC
NSENT=SWVAR EOLC
```

```
NSLIST=NSENT
NSLIST=NSLIST NSENT
NUMCARD=NUMBER
NUMCARD=NUMCARD$,$NUMBER
NUMLIST=NUMCARD EOLC
NUMLIST=NUMLIST NUMCARD EOLC
OBJ=KW34$=$ME AOBJ
OBJ=KW34 $=$ ME
OPTENT=GRP3
OPTENT=GRP1
OPTENT=GRP4
OPTENT=CURVE
OPTENT=STATECARD EOLC STATELIST STATEEND
OPTLIST=OPTENT
OPTLIST=OPTLIST OPTENT
OUT=KW10 POUTLST
OUT=KW19 $,$ KW20 POUTLST
OUT=KW19 POUTLST
OUTLST=ME AOUTLST
OUTLST=ME
OUTLST=OUTLST $,$ ME AOUTLST
OUTLST=OUTLST $,$ ME
OUTVAR=TDRV
OUTVAR=FDRV
OUTVAR=ELVAL
OUTVAR=TDVAR
OUTVAR=KW49 $($ TDVAR $)$
OUTVAR=KW32 $($ TDVAR $)$
OUTVAR=KW1 $($ DEPVARLST $/$ INDVARLST $)$
PARAM=KW7 $=$ KW22
PARAM=INDVAR$=$APARAM
POUTLST=$,$ OUTLST $,$ $/$ ME
POUTLST=$,$ OUTLST $,$ $/$ ME AOUTLST
POUTLST=$,$ OUTLST
RC=KW12$=$INTEGER
RC=KW13$=$ME
RC=KW14$=$KW15
RC=KW16
RC=KW17 $($ BEXP $)$
RC=KW18$=$NUMBER
REALVAL=REALNAME
REALVAL=NEST$,$ REALNAME
RUNEND=$END$ EOLC
RUNEND=ENDRUN EOLC
STATEEND=$END$ EOLC
STATEEND=ENDSTATE EOLC
STATELIST=STATENT
STATELIST=STATELIST STATENT
STATENT=GRP1
STATENT=GRP4
STATENT=RC EOLC
STATENT=OBJ EOLC
STATENT=OUTVAR EOLC
```

```
SUBNETCARD=KW29 $,$ ILL $,$ DUMNODELIST $=$ ARGLIST
SUBNETCARD=KW29 $,$ ILL $,$ DUMNODELIST
SUBNETLIST=GRP1
SUBNETLIST=SUBNETLIST GRP1
SUBNETLIST=SUBNETLIST GRP2
SUBNETLIST=GRP2
SUBNETREF=SUBNETNAME $,$ NODELIST $=$ SUBNETVAL
SUBNETREF=SUBNETNAME $,$ NODELIST
SUBNETVAL=VALUELIST1
SUBNETVAL=$$$$ IAA
SUBNETVAL=$$$$ IAA $($ VALUELIST2 $)$
SWNUMLIST=$,$ BSWNUMLIST ASWNUM
SWNUMLIST=$,$ ASWNUM
SWVAR=PARAM SWNUMLIST
SWVAR=PARAM SWNUMLIST $,$ KW22
SWVAR=PARAM $,$ KW22
TABLE=KW25 $,$ KW26 EOLC NUMLIST
TABLE=KW25 EOLC NUMLIST
TABLE=KW25 EOLC NUMLIST NUMBER $,$ KW27 EOLC
TARGLANG=$TARGET$ $=$ LANG EOLC
TDRV=KW2 $($ NODE $)$
TDRV=KW4 $($ ELNAME $)$
TDRV=KW50 $($ ELNAME $)$
TDVAR=KW7
TDVAR=REALVAL
VALUELIST1=ME
VALUELIST1=VALUELIST1$,$ME
VALUELIST2=KW6$=$ME
VALUELIST2=VALUELIST2$,$KW6$=$ME
VALUES=KW9 $($ ME $,$ ME $)$
VALUES=$($ ME $,$ ME $)$
VALUES=VALUELIST1
WCENT=GRP1
WCENT=GRP4
WCENT=GRP3
WCENT=KW10 $,$ OUTLST EOLC
WCLIST=WCENT
WCLIST=WCLIST WCENT
/END

**
```

APPENDIX F

CSL LEXICAL ANALYZER DESCRIPTION

This appendix lists the input to the Lexical Analyzer Generator which produces the lexical analyzer tables to be used with the parser for CSL described in Appendix E. The description of the lexical analyzer is:

```
/COMMENT LEXICAL ANALYZER FOR CSL GRAMMAR 7/25/75
/COMMENT RUN CONTROLS AND MOST PARAMETER SYMBOLS MISSING
/MAXSCAN 12
/WORKSPACE 1
/HASHTABLE 751
/IDTABLE 2000
/CHARTABLE 49
/CHARCODE
A 1
B 2
C 3
D 4
E 5
F 6
G 7
H 8
I 9
J 10
K 11
L 12
M 13
N 14
O 15
P 16
Q 17
R 18
S 19
T 20
U 21
V 22
W 23
X 24
Y 25
Z 26
D0 27
D1 28
D2 29
D3 30
D4 31
D5 32
```

```
D6 33
D7 34
D8 35
D9 36
PLUS 37
MINUS 38
STAR 39
SLASH 40
LP 41
RP 42
DOLLAR 43
EQ 44
BLANK 45
COMMA 46
DOT 47
EOLC 48
NOCARD 49
/LEXCODE
FINAL 92
LINLOG 90
LOGLIN 90
LOGLOG 90
POLAR 90
NYQUIST 90
COMMENT 129
TEXTLIST 130
EQ 17
COMMA 63
STAR 64
NUMBER 53
EXPONENT 65
USETEXT 81
DOT 18
SLASH 14
INTEGER 2
RP 16
PLUS 50
MINUS 67
EOLC 10
DOLLAR 134
/SCANNER 1
/COMMENT KEYWORDS NOT INVOLVING DOT OR STARTING LINE, TYPE ILA
/COMMENT CURVE(93) SENT TO SCANNER 11, RECOGNIZED BY FSA
/CLASS
3 A B D E F G H I J K L M N O P Q R S T U V W X Y Z
5 EQ LP RP EOLC COMMA
5 C
6 BLANK
8 PLUS MINUS DOT
8 D0 D1 D2 D3 D4 D5 D6 D7 D8 D9
9 A B C D E F G H I J K L M N O P Q R S T U V W X Y Z
9 D0 D1 D2 D3 D4 D5 D6 D7 D8 D9
```

```
/FSA
$1,EOLC,EOLC,$2
$2,COMMA,COMMA,$3
$3,EQ,EQ,$5
$5,LP,LP,$6
$6,RP,RP,$7
$7,C,$8,1
$8,U,$9,3
$9,R,$10,3
$10,V,$11,3
$11,E,$12,3
$12,8,*11,3
/KEYWORDS
X 105
TWOD 96
SCEPTRE 127
NET 127
CIRCUS 127
LOG 91
R 97
TARGET 126
FINAL
LINLOG
LOGLIN
LOGLOG
POLAR
NYQUIST
GAUSS 103
RECT 103
TRI 103
/SCANNER 2
/COMMENT FIXED LENGTH KEYWORDS AND SPECIAL CHARACTERS
/COMMENT IDENTIFIER (TYPE ILA) IS FORMAL PARAMETER IN DEFINITIONS OF
/COMMENT SUBNET, MODEL, FUNCTION, PROCEDURE
/IDENTIFIER 52
/CLASS
4 A B C D E F G H I J K L M N O P Q R S T U V W X Y Z
2 DOT
5 PLUS MINUS RP DOLLAR EOLC COMMA SLASH EQ
5 LP
6 BLANK
8 STAR
9 A B C D E F G H I J K L M N O P Q R S T U V W X Y Z DOT
9 D0 D1 D2 D3 D4 D5 D6 D7 D8 D9
/KWLENGTH 1 4 5
/KEYWORDS
.LT. 61
.LE. 61
.GT. 61
.GE. 61
.EQ. 61
.NE. 61
. 18
```

```
.NOT. 62
.OR. 59
.AND. 60
/FSA
$1,PLUS,PLUS,$2
$2,MINUS,MINUS,$3
$3,SLASH,SLASH,$4
$4,COMMA,COMMA,$5
$5,EQ,EQ,$6
$6,RP,RP,$7
$7,EOLC,EOLC,$8
$8,DOLLAR,DOLLAR,$9
$9,LP,LP,1
/SCANNER 3
/COMMENT KEYWORDS WITHOUT DIGITS OR DOTS, STARTING LINE, ALSO COMMENT
/COMMENT LEXICAL CONSTRUCTS WHICH MAY START A LINE
/COMMENT CLASS 8 RECOGNIZES NUMBER, CURVE, TABLE, STANDARD
/COMMENT ELEMENTS, MODEL REFERENCE, SUBNET REFERENCE, COMMENT
/COMMENT (I.E., ANY ILL IDA COMMA CONSTRUCT AND CERTAIN ILL IDA EOLC
/COMMENT CONSTRUCTS)
/COMMENT SCANNER 3 IS ENTERED FOR REAL VARIABL DEFINITION AND KEYWORDS
/COMMENT (ILA EQ CONSTRUCT) BUT ANALYSIS DONE IN SCANNER 6 VIA
/COMMENT FSA TRANSFER
/COMMENT SCANNER 3 IS ENTERED FOR FUNCTION DEFINITION AND KEYWORDS
/COMMENT (ILA LP CONSTRUCT) BUT ANALYSIS IS DONE IN SCANNER 10 VIA
/COMMENT FSA TRANSFER
/COMMENT KEYWORDS TEXT, USETEXT, TABLE, CURVE, RECOGNIZED BY FSA. TABLE
/COMMENT AND CURVE THEN PROCESSED BY CLASS 8 ROUTINE.
/CLASS
5 A B C D E F G H I J K L M N O P Q R S T U V W X Y Z
5 STAR
8 D0 D1 D2 D3 D4 D5 D6 D7 D8 D9 PLUS MINUS DOT DOLLAR
6 BLANK
9 A B C D E F G H I J K L M N O P Q R S T U V W X Y Z
9 D0 D1 D2 D3 D4 D5 D6 D7 D8 D9
/FSA
$1,T,$41,$27
$14,STAR,STAR,$15
$15,5,$15,$16
$16,COMMA,3,$17
$17,LP,*10,$18
$18,EOLC,3,$19
$19,EQ,*6,$20
$20,8,$21,1
$21,5,$21,$22
$22,8,$21,$23
$23,EOLC,3,$24
$24,EQ,*6,$25
$25,LP,*10,$26
$26,COMMA,8,1
$27,U,$34,$28
$28,C,$29,$14
$29,U,$30,$15
```

```
$30,R,$31,$15
$31,V,$32,$15
$32,E,$33,$15
$33,8,$45,$15
$34,S,$35,$15
$35,E,$36,$15
$36,T,$37,$15
$37,E,$38,$15
$38,X,$39,$15
$39,T,$40,$15
$40,8,3,$15
$41,A,$42,$37
$42,B,$43,$15
$43,L,$44,$15
$44,E,$33,$15
$45,5,$45,$46
$46,8,$45,$47
$47,EOLC,8,$24
/KEYWORDS
PRINT 80
ENDRUN 109
RUN 117
STATE 115
DATUM 118
SUBNET 99
ENDENTER 121
ENTERCIRCUS 94
ENTERNET 94
ENTERSCEPTRE 94
ENDSTATE 116
ENDDECK 107
RUNMONTECARLO 108
RUNOPTIMIZATION 114
MODEL 123
PLOT 89
BOUNDS 113
ENDMODEL 124
END 69
RUNWORSTCASE 110
DEVICE 125
INITIALCONDITIONS 111
ENDSUBNET 112
PROCEDURE 98
/SCANNER 4
/COMMENT FIRST SYMBOL OF ME
/COMMENT CLASS 8 ROUTINE HANDLES SIGNED NUMBERS (53), PLUS(50),MINUS(67)
/COMMENT KEYWORDS BEGINNING WITH DOT ARE ANALYZED BY SCANNER 2 THRU
/COMMENT FSA XFR
/COMMENT FSA RECOGNIZES LP(15), SLASH(14), COMMA(63), EOLC(10),
/COMMENT DOLLAR(134)
/COMMENT KW1(1), MATH FUNC(122), USER FUNC(135), KW9(79), KW32(102),
/COMMENT KW49(119), KW50(120), KW2(72), KW4(74) SENT TO SCANNER 10
```

```
/COMMENT REAL VARIABLE(132) (TYPE ILL), KW7(77) (TYPE ILL), KW20(90),
/COMMENT KW22(92), KW33(103) SENT TO SCANNER 6
/COMMENT REAL VARIABLE (132) (TYPE ILA), STANDARD ELEMENT (78),
/COMMENT KW7 (77) (TYPE ILA),
/COMMENT SUBNET REFERENCES (131), CURVE (93) SENT TO SCANNNER 11
/CLASS
6 BLANK
5 A B C D E F G H I J K L M N O P Q R S T U V W X Y Z
5 DOT LP  SLASH COMMA EOLC DOLLAR
8 D0 D1 D2 D3 D4 D5 D6 D7 D8 D9
8 PLUS MINUS
/FSA
$1,SLASH,SLASH,$4
$2,8,8,*2
$3,LP,LP,$16
$4,COMMA,COMMA,$5
$5,EOLC,EOLC,$6
$6,DOLLAR,DOLLAR,$7
$7,DOT,$2,$3
$8,RP,*6,$9
$9,SLASH,*6,$10
$10,COMMA,*6,$11
$11,EOLC,*6,$12
$12,DOLLAR,*6,$14
$13,PLUS,*6,$27
$14,LP,*10,$15
$15,DOT,*6,$16
$16,5,$8,$13
$17,LP,*10,$18
$18,DOT,*11,$19
$19,SLASH,*11,$20
$20,DOLLAR,*11,$21
$21,COMMA,*11,$22
$22,EOLC,*11,$23
$23,PLUS,*11,$24
$24,MINUS,*11,$25
$25,5,$17,$26
$26,8,$17,*11
$27,MINUS,*6,$28
$28,8,$17,$29
$29,LP,*10,*6
/SCANNER 5
/COMMENT ILL TYPE
/IDENTIFIER 55
/CLASS
4 A B C D E F G H I J K L M N O P Q R S T U V W X Y Z
6 BLANK
9 A B C D E F G H I J K L M N O P Q R S T U V W X Y Z
/SCANNER 6
/COMMENT RECOGNIZES KEYWORDS AND REAL VARIABLE DEFINITION (LEXCODE=132)
```

```
/COMMENT RECEIVES CONTROL BY FSA XFR FROM SCANNERS 3 AND 4
/CLASS
3 A B C D E F G H I J K L M N O P Q R S T U V W X Y Z
6 BLANK
9 A B C D E F G H I J K L M N O P Q R S T U V W X Y Z
9 D0 D1 D2 D3 D4 D5 D6 D7 D8 D9
/IDENTIFIER 132
/KEYWORDS
OBJ 104
TIME 77
FREQ 77
DELTAT 77
NEUT 77
GAMMA 77
GAMDOT 77
MAXSTEP 77
MINSTEP 77
TERMINATE 77
LINLOG
LOGLIN
LOGLOG
POLAR
NYQUIST
FINAL
/SCANNER 7
/COMMENT NODE NAME PROCESSING AND TYPE IAA CONSTRUCTS
/COMMENT ELEMENT NAMES(78) AND SUBNET NAMES(131) SENT TO SCANNER 11
/IDENTIFIER 54
/CLASS
5 A B C D E F G H I J K L M N O P Q R S T U V X Y Z
5 SLASH
4 D0 D1 D2 D3 D4 D5 D6 D7 D8 D9
6 BLANK
9 A B C D E F G H I J K L M N O P Q R S T U V W X Y Z
9 D0 D1 D2 D3 D4 D5 D6 D7 D8 D9
/FSA
$1,SLASH,SLASH,$2
$2,4,4,$3
$3,SLASH,4,$4
$4,5,$3,$5
$5,4,$3,$6
$6,DOT,*11,4
/SCANNER 8
/COMMENT ELEMENT PARAMETER SYMBOLS
/COMMENT ONLY DIODE PARAMETERS INCLUDED
/CLASS
3 A B C D E F G H I J K L M N O P Q R S T U V W X Y Z
6 BLANK
9 A B C D E F G H I J K L M N O P Q R S T U V W X Y Z
9 D0 D1 D2 D3 D4 D5 D6 D7 D8 D9
/KEYWORDS
C 76
GC 76
```

```
IP1 76
IP2 76
IS 76
KIS 76
KRB 76
KW 76
N 76
RB 76
T 76
TH 76
VZ 76
W 76
/SCANNER 9
/COMMENT TEXTLIST (TERMINATES WITH ENDTEXT OR ENDENTER)
/CLASS
8 A B C D E F G H I J K L M N O P Q R S T U V W X Y Z
8 D0 D1 D2 D3 D4 D5 D6 D7 D8 D9
8 PLUS MINUS STAR SLASH LP RP DOLLAR EQ BLANK
8 COMMA DOT EOLC
/SCANNER 10
/COMMENT RECOGNIZES KEYWORDS AND FUNCTION DEFINITIONS (LEXCODE=133)
/COMMENT RECEIVES CONTROL BY FSA XFR FROM SCANNERS 3 AND 4
/COMMENT HANDLES ILA LP CONSTRUCTS
/IDENTIFIER 135
/CLASS
5 A B C D E F G H I J K L M N O P Q R S T U V W X Y Z
6 BLANK
9 A B C D E F G H I J K L M N O P Q R S T U V W X Y Z
9 D0 D1 D2 D3 D4 D5 D6 D7 D8 D9
/KWLENGTH 1
/KEYWORDS
LOG 122
SQRT 122
TAN 122
U 122
ABS 122
ATAN 122
ATAN2 122
COS 122
EXP 122
MAX 122
MIN 122
MOD 122
SIGN 122
SIN 122
COMPLEX 79
RADIANS 79
DEGREES 79
SENS 71
PT 119
PV 119
IV 119
D 119
S 102
```

```
Q 120
E 120
F 120
N 72
A 75
B 75
Y 75
Z 75
I 74
V 74
P 74
/FSA
$1,I,$10,$2
$2,V,$10,$3
$3,P,$10,$4
$4,A,$10,$5
$5,B,$10,$6
$6,Y,$10,$7
$7,Z,$10,$8
$8,N,$9,3
$9,LP,2,$10
$10,M,$15,$11
$11,D,$15,$12
$12,P,$15,$13
$13,R,$15,$14
$14,I,$15,3
$15,LP,2,3
/SCANNER 11
/COMMENT RECOGNIZES REALNAME, STANDARD ELEMENT, SUBNET REFERENCES,
/COMMENT CURVE USING CLASS 8 ROUTINE
/COMMENT RECEIVES CONTROL BY FSA XFR FROM SCANNERS 4 AND 7
/CLASS
6 BLANK
8 A B C D E F G H I J K L M N O P Q R S T U V W X Y Z
9 A B C D E F G H I J K L M N O P Q R S T U V W X Y Z
9 D0 D1 D2 D3 D4 D5 D6 D7 D8 D9
/KEYWORDS
CURVE 93
TABLE 95
ABS 78
AND 78
ACOS 78
ASIN 78
ATAN 78
C 78
CONV 78
CW 78
D 78
DDACC 78
DELAY 78
DERIV 78
DF 78
EOR 78
EXP 78
```

```
EXPN 78
GAIN 78
HC 78
HD 78
HYST 78
I 78
INPFL 78
INT 78
IPP 78
JFET 78
K 78
L 78
LIM 78
LIMINT 78
LOG 78
MAX 78
MC 78
MFET 78
MIN 78
MOD 78
MULT 78
NLVCCS 78
NORM 78
OR 78
OUTFL 78
PN 78
QUANT 78
R 78
RADC 78
RMS 78
RNGEN 78
RSTFF 78
S 78
SAMPL 78
SIGN 78
SINCOS 78
SNCLK 78
SQRT 78
ST 78
SUM 78
T 78
TABF 78
TANH 78
TD 78
TLINE 78
V 78
VCCS 78
VCG 78
VCVS 78
XFCP 78
XFCZCP 78
XFCZDP 78
XFP 78
```

```
XFSP 78
XFZCP 78
ZD 78
XMOD 78
YMOD 78
/SCANNER 12
/COMMENT INTEGER AND KW5A(128)
/CLASS
2 M D P R I
5 LP
6 BLANK
8 D0 D1 D2 D3 D4 D5 D6 D7 D8 D9
/KWLENGTH 1
/KEYWORDS
M 128
P 128
R 128
I 128
D 128
/FSA
$1,LP,LP,1
/END
**
```

APPENDIX G

NET-2 GRAMMAR

This appendix lists the grammar for the NET-2 language using BNF notation which is consistent with the input to the LR(k) Parser Generator. A listing of the undefined terminal symbols and their corresponding character strings is given below:

Lexical Symbol	Character String
ACPFX	A, A', B, B', Y, Y', Z, or Z'
ARG	Any alphameric identifier, starting with a letter, which may serve as a formal parameter
B	One or more blanks
CIRCPFX	Any of the standard circuit element prefixes, not including the modeled devices
DEVPFX	Any of the modeled device prefixes
DRTYPE	Any of the data reduction types, e.g., DC, CT, etc.
EOLC	End of logical card symbol
GLOBAL	Any global variable, e.g., FREQ, TIME, TERMINATE, etc.
ILL	Any identifier composed only of letters
INTEGER	Any unsigned integer
MATHF	Any standard mathematical function, e.g., SIN, COS, etc.
NUMBER	Any real number representation
PARAM	Any device parameter
PRIME	
RPFX	Any element response variable prefix, e.g., I, V, P, etc.
SINT	An integer with a mandatory + or - sign prefix
SUFFIX	Any alphameric identifier, starting with a digit
SYSPFX	Any system element prefix
T	A tab or indentation, consisting of a fixed number of blanks
TEXT	An arbitrary character string
TYPE	Any alphameric identifier, representing a device type name

The complete description of the NET-2 grammar is:

```
/COMMENT NET-2 GRAMMAR
/DEBUG 18
/LEXICAL
T 10
B 11
EOLC 12
ACPFX 13
ARG 14
CIRCPFX    16
DEVPFX     17
DRTYPE     18
GLOBAL     19
INTEGER    20
NUMBER     21
PARAM 22
RPFX  23
SUFFIX     24
SINT  25
SIMPNODE 26
SYSPFX     27
TEXT  28
TYPE  29
WORD  30
$END$ 31
$+$ 32
$-$ 33
$,$ 34
$.$ 35
$MODE$ 36
$K$ 37
$L$ 38
$CW$ 39
$MC$ 40
$*$ 41
$=$ 42
$FREQ$ 43
$TERMINATE$ 44
$MAXSTEPS$ 45
$DEBUG$ 46
$TLIM$ 47
$CURVE$ 48
$DATA$ 49
$DEFINE$ 50
$ALL$ 51
$DIST$ 52
$DELETE$ 53
$DPL$ 54
```

```
$PRINT$ 55
$LP$ 56
$LD$ 57
$($ 58
$)$ 59
$F$ 60
$COPY$ 61
$LIBRARY$ 62
$SML$ 63
$MONTECARLO$ 64
$TIME$ 65
$PLOT$ 67
$**$ 68
$MODEL$ 69
$OFF$ 70
$NONE$ 71
$/$ 72
$OBJ$ 73
$OPTIMIZE$ 74
$STATE$ 75
$VS$ 76
$PARAMETER$ 77
$P$ 78
$LINLOG$ 79
$LOGLIN$ 80
$LOGLOG$ 81
$POLAR$ 82
$N$ 83
$GAUSS$ 84
$RECT$ 85
$TRI$ 86
$X$ 87
PRIME 88
$R$ 89
$FINAL$ 90
MATHF 91
$CASES$ 92
$TABLE$ 93
/BNF
RUN=DECK $END$ EOLC
AOP=$+$
AOP=$-$
ARGLIST=ARG
ARGLIST=ARGLIST $,$ ARG
CASE=$CASES$ B INTEGER EOLC
CHANGE1=ELVAL B NUM2 EOLC
CHANGE1=DEVNAME $,$ $MODE$ B MODE1 EOLC
CIRCELEM=CIRCPFX SUFFIX B PARSEQ B NODELIST B VALUELIST EOLC
CIRCELEM=CIRCPFX SUFFIX B NODELIST B VALUELIST EOLC
CIRCELEM=$K$ SUFFIX B $L$ SUFFIX B $L$ SUFFIX B VALUE EOLC
CIRCELEM=$CW$ SUFFIX B NODELIST B $MC$ SUFFIX B VALUELIST EOLC
COMMENT=$*$ TEXT EOLC
CONSTRAINT=ELVAL $=$ ME EOLC
CONSTRAINT=$FREQ$ $=$ ME EOLC
```

```
CONTROL=$TERMINATE$ $=$ ME EOLC
CONTROL=$MAXSTEP$ $=$ ME EOLC
CONTROL=$DEBUG$ B INTEGER EOLC
CONTROL=$TLIM$ B NUMBER EOLC
CURV=$CURVE$ SUFFIX EOLC T NUMBER B NUM1 EOLC
CURV=CURV T NUMBER B NUM1 EOLC
DATA1=T T $DATA$ B DRTYPE EOLC
DATA1=T T $DATA$ B DRTYPE B PARAM EOLC
DATA1=DATA1 T T T NUMSET EOLC
DECK=ENTRY
DECK=DECK ENTRY
DEFENT=SUBNETREF
DEFENT=ELEMENT
DEFENT=XDEF
DEFENT=SYMCON
DFINE=$DEFINE$ B SUBNETPFX B DUMNODELIST EOLC T DEFENT
DFINE=DFINE T DEFENT
DEVICE=DEVPFX SUFFIX B PARSEQ B NODELIST B TYPE1 EOLC
DEVICE=DEVPFX SUFFIX B NODELIST B TYPE1 EOLC
DEVLIST=T T DEVPFX B TYPE EOLC
DEVLIST=DEVLIST T T DEVPFX B TYPE EOLC
DEVLIST1=DEVLIST
DEVLIST1=T T $ALL$ EOLC
DEVNAME=NEST $.$ DEVPFX SUFFIX
DEVNAME=DEVPFX SUFFIX
DISTDEF=$DIST$ SUFFIX EOLC T NUM1 EOLC
DISTDEF=DISTDEF T NUM1 EOLC
DLIBDEL=T $DELETE$ B $DPL$ EOLC DEVLIST
DLIBPRINT=T $PRINT$ B $DPL$ B INTEGER EOLC DEVLIST1
DLIBPRINT=T $PRINT$ B $DPL$ EOLC DEVLIST1
DUMNODELIST=SIMPNODE
DUMNODELIST=DUMNODELIST B SIMPNODE
ELEM=ELPFX SUFFIX
ELEMENT=CIRCELEM
ELEMENT=SYSELEM
ELEMENT=DEVICE
ELNAME=ELEM
ELNAME=NEST $,$ ELEM
ELPFX=CIRCPFX
ELPFX=DEVPFX
ELPFX=SYSPFX
ELPTR=ELPTR1
ELPTR=ELPTR1 SINT
ELPTR1=$L$
ELPTR1=$LP$
ELPTR1=$LD$
ELVAL=ELNAME
ELVAL=ELNAME $.$ INTEGER
ELVAL=ELNAME $.$ PARAM
ELVAL=ELNAME $.$ $($ ELPTR $)$
ENTRY=DFINE
ENTRY=SUBNETREF
ENTRY=ELEMENT
ENTRY=$FREQ$ B NUMBER EOLC
```

```
ENTRY=SOLN
ENTRY=CONTROL
ENTRY=COMMENT
ENTRY=TABEL
ENTRY=CURV
ENTRY=DISTDEF
ENTRY=XDEF
ENTRY=SYMCON
ENTRY=FUNC
ENTRY=PRMETER
ENTRY=LIB
ENTRY=MODELDEF
FUNC=$F$ SUFFIX $($ ARGLIST $)$ $=$ ME EOLC
LIBCOPY=T $COPY$ B LIBTYPE EOLC
LIBENT=DLIBPRINT
LIBENT=LIBSPEC
LIBENT=DLIBDEL
LIBENT=LIBCOPY
LIBENT=SLIBDEL
LIBENT=SLIBPRINT
LIB=$LIBRARY$ EOLC LIBENT
LIB=LIB LIBENT
LIBSPEC=T DEVPFX B TYPE B INTEGER EOLC
LIBSPEC=LIBSPEC T T PARAM B NUMBER
LIBSPEC=LIBSPEC T T DATA1
LIBTYPE=$DPL$
LIBTYPE=$SML$
MARG=OUTVAR
MARG=NUMBER
MARG=ARG
MC1=$MONTECARLO$ INTEGER EOLC T MCENT
MC1=MC1 T MCENT
MCENT=CHANGE1
MCENT=STAT
MCENT=CASE
MCENT=$TIME$ B NUM2 EOLC
MCENT=$FREQ$ B NUM2 EOLC
MCENT=MCOUT
MCOUT=$PRINT$ B OUTLST EOLC
MCOUT=$PLOT$ B OUTVAR EOLC
ME=ADP MTERM
ME=MTERM
ME=ME ADP MTERM
MFACTOR=MFACTOR1
MFACTOR=MFACTOR1 $**$ MFACTOR1
MFACTOR1=$($ ME $)$
MFACTOR1=MARG
MFACTOR1=MFUNC $($ VALUELIST $)$
MFUNC=MATHF
MFUNC=$F$ SUFFIX
MODELDEF=$MODEL$ B SUBNETPFX B DUMNODELIST EOLC T MODELENT
MODELDEF=MODELDEF T MODELENT
MODELENT=SUBNETREF
```

```
MODELENT=ELEMENT
MODELENT=XDEF
MODELENT=SYMCON
MODELENT=TABEL
MODELENT=FUNC
MODELENT=DFINE
MODE1=$OFF$
MODE1=$NONE$
MOP=$*$
MOP=$/$
MTERM=MFACTOR
MTERM=MTERM MOP MFACTOR
NEST=SUBNET
NEST=NEST $.$ SUBNET
NODE=SIMPNODE
NODE=NEST $.$ SIMPNODE
NODELIST=NODE
NODELIST=NODELIST B NODE
NUMSET=NUMBER B NUMBER
NUMSET=NUMSET B NUMBER
NUM1=NUMBER
NUM1=NUM1 B NUMBER
NUM2=NUMBER
NUM2=NUMBER $*$
OBJCT=$OBJ$ $=$ ME EOLC
OBJCT=$OBJ$ $=$ ME OBJ1 EOLC
OBJ1=B $CURVE$ SUFFIX B NUMBER
OBJ1=B $CURVE$ SUFFIX B NUMBER B $X$
OPT=$OPTIMIZE$ INTEGER EOLC T OPTENT
OPT=OPT T OPTENT
OPTENT=RANGE
OPTENT=CONSTRAINT
OPTENT=OPTSTATE
OPTSTATE=$STATE$ EOLC T T OPTSTATENT
OPTSTATE=OPTSTATE T T OPTSTATENT
OPSTATENT=CHANGE1
OPTSTATENT=ELVAL B SWNUM EOLC
OPTSTATENT=$TIME$ B SWNUM EOLC
OPTSTATENT=$FREQ$ B SWNUM EOLC
OPTSTATENT=PCON B SWNUM EOLC
OPTSTATENT=OBJCT
OUT=T $PRINT$ OUTLST EOLC
OUT=PLOT1 B OUTLST EOLC
OUT=PLOT1 B OUTLST B $VS$ B OUTVAR EOLC
OUTLST=OUTVAR
OUTLST=OUTLST B OUTVAR
OUTVAR=ELVAL
OUTVAR=RVAR
OUTVAR=GLOBAL
OUTVAR=XVARNAME
OUTVAR=PCON
PRMETER=$PARAMETER$ EOLC T CHANGE1
PRMETER=PRMETER T CHANGE1
```

```
PARSEQ=$($ INTEGER $)$
PCON=$P$ SUFFIX
PLOTTYPE=$LINLOG$
PLOTTYPE=$LOGLIN$
PLOTTYPE=$LOGLOG$
PLOTTYPE=$POLAR$
PLOT1=$PLOT$
PLOT1=$PLOT$ B PLOTTYPE
RANGE=RANGE1 B NUM2 B NUM2 EOLC
RANGE1=ELVAL
RANGE1=$FREQ$
RANGE1=PCON
RVAR=$N$ $($ NODE $)$
RVAR=ACPFX $($ NODE $-$ NODE $/$ NODE $-$ NODE $)$
RVAR=RPFX $($ ELNAME $)$
SLIBDEL=T $DELETE$ B $SML$ EOLC T T SUBNETPFX EOLC
SLIBDEL=SLIBDEL T T SUBNETPFX EOLC
SLIBPRINT=T $PRINT$ B $SML$ B INTEGER EOLC T T SUBNETPFX EOLC
SLIBPRINT=T $PRINT$ B $SML$ EOLC T T SUBNETPFX EOLC
SLIBPRINT=SLIBPRINT T T SUBNETPFX EOLC
SOLN=STATE1
SOLN=MC1
SOLN=OPT
STAT=STAT1 B STAT2 NUM2 EOLC
STATDELIM=B
STATDELIM=$*$ B
STATE1=$STATE$ INTEGER EOLC T STATENT
STATE1=STATE1 T STATENT
STATENT=T CHANGE1
STATENT=T STATENT1 B SWNUM EOLC
STATENT=T $*$ STATENT1 B SWNUM EOLC
STATENT=OUT
STATENT=$TIME$ B SWNUM B $FINAL$ EOLC
STATENT=$TIME$ B $FINAL$ EOLC
STATENT1=ELVAL
STATENT1=$TIME$
STATENT1=$FREQ$
STATENT1=PCON
STAT1=ELVAL
STAT1=$TIME$
STAT1=$FREQ$
STAT1=PCON
STAT2=$GAUSS$ STATDELIM
STAT2=$RECT$ STATDELIM NUM2 B
STAT2=$TRI$ STATDELIM NUM2 B
STAT2=$DIST$ SUFFIX STATDELIM NUM2 B
SUBNET=SUBNETPFX SUFFIX
SUBNETPFX=WORD
SUBNETREF=SUBNET B PARSEQ B NODELIST EOLC
SUBNETREF=SUBNET B NODELIST EOLC
SWNUM=NUM2
SWNUM=SWNUM B NUM2
SWNUM=SWNUM $($ INTEGER $)$ NUM2
SWNUM=SWNUM $($ INTEGER $*$ $)$ NUM2
```

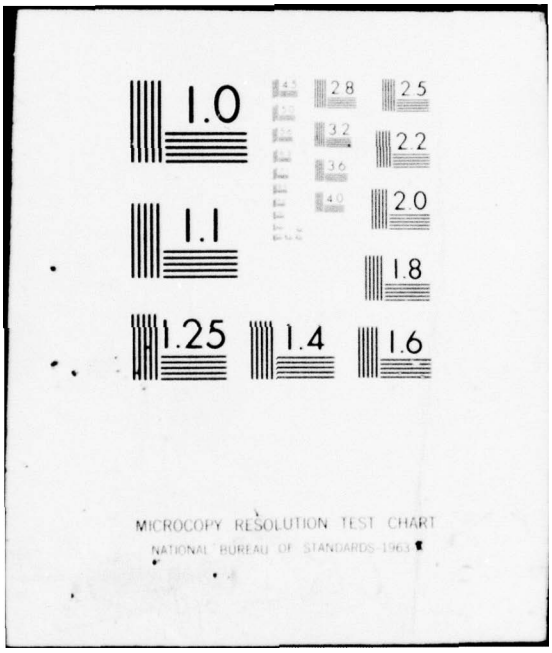
```
SYMCON=$P$ SUFFIX B NUMBER EOLC
SYMLIST=OUTLST
SYMLIST=$TABLE$ SUFFIX
SYMLIST=$MC$ SUFFIX
SYSELEM=SYSPFX SUFFIX EOLC
SYSELEM=SYSPFX SUFFIX SYS1 EOLC
SYS1=B NODELIST
SYS1=B NODELIST SYS2
SYS1=SYS2
SYS2=B VALUELIST
SYS2=B VALUELIST B SYMLIST
SYS2=B SYMLIST
TABEL=TABLE1
TABEL=TABEL T NUMBER B $R$ EOLC
TABEL=TABLE2
TABLE1=$TABLE$ SUFFIX EOLC T NUMBER B NUMBER EOLC
TABLE1=TABLE1 T NUMBER B NUMBER EOLC
TABLE2=$TABLE$ SUFFIX B INTEGER EOLC T NUMSET EOLC T NUMSET EOLC
TABLE2=TABLE2 T NUMSET EOLC
TYPE1=TYPE
TYPE1=TYPE B MODE
TYPE1=TYPE B NUMBER
VALUE=NUMBER
VALUE=ME
VALUELIST=VALUE
VALUELIST=VALUELIST $,$ VALUE
XDEF=$X$ SUFFIX $=$ ME EOLC
XDEF=$X$ PRIME SUFFIX $=$ ME EOLC
XVAR=$X$ SUFFIX
XVAR=$X$ PRIME SUFFIX
XVARNAME=XVAR
XVARNAME=NEST $.$ XVAR
/END
""
```

AD-A033 296 BDM CORP EL PASO TEX F/G 9/2
ON THE DESIGN OF AN EXECUTIVE PROGRAM AND A COMMON SIMULATION L--ETC(U)
SEP 76 A F MALMBERG F29601-74-C-0017
UNCLASSIFIED BDM/E-47-75-F-0017 AFWL-TR-75-272 NL

3 OF 3
AD
A033296

END

DATE
FILMED
2-77


1.0
1.1
1.25
1.4
1.6
2.8
3.2
3.6
4.0
2.5
2.2
2.0
1.8
MICROCOPY RESOLUTION TEST CHART
NATIONAL BUREAU OF STANDARDS-1963

APPENDIX H

SCEPTRE GRAMMAR

This appendix lists the grammar for the SCEPTRE language using BNF notation which is consistent with the input to the LR(k) Parser Generator. A listing of the undefined terminal symbols and their corresponding character strings is given below:

Lexical Symbol	Character String
AND	.AND.
AOP	+ or -
CONSTANT	A SCEPTRE Constant
DERIV	D or DERIVATIVE
EOLC	End of logical card symbol
IAA	Any alphameric identifier
ILA	Any alphameric identifier starting with a letter
INTEGER	Any unsigned integer
INTPAR	Any internal parameter name, e.g., XSTOPT, XIR, etc.
IRA	Any alphameric identifier, beginning with the letters A through H or O through Z
KWDTABLE	T, TABLE, or DIODE TABLE
KWEQN	Q, EQUATION, or DIODE EQUATION
KWEXP	X or EXPRESSION
KWG2	R, L, C, E, or J
KWG2A	V or I
KWG2B	DE, DJ, or M
KWG4	P or M
KWG5	J, P, VR, VC, VJ, IR, IL, or IE
KWG6	R, E, J, or P
KWG7C	VC or IL
KWG7D	R, E, or J
KWICSH	VC, VJ, or IL
KWMODOPS	PERM, TEMP, or PRINT
KWOUT	DEGREES, RADIANS, COMPLEX, or NYQUIST

Lexical Symbol	Character String
KWRC1	A run control identifier
KWRC2	A run control identifier
KWRC3	A run control identifier
KWRC4	A run control identifier
KWRC5	A run control identifier
KWRC6	A run control identifier
KWRC7	A run control identifier
KWRRC1	A run control identifier under RERUN
KWRRC2	A run control identifier under RERUN
KWRRC3	A run control identifier under RERUN
KWRRC4	A run control identifier under RERUN
KWRRC6	A run control identifier under RERUN
KWRRC7	A run control identifier under RERUN
KWTABLE	T or TABLE
KWVAR	R, C, L, E, J, M, DE, or DJ
MFUNC	Any standard mathematical function, e.g., SIN, COS, etc.
MOP	* or /
NOT	.NOT.
NUMBER	A SCEPTRE Number
OR	.OR.
RELOP	.EQ., .NE., .GT., .GE., .LT., or .LE.
REMARK	An arbitrary character string

The complete description of the SCEPTRE grammar except for features associated with CONTINUE and RE-OUTPUT is:

```
/COMMENT SCEPTRE GRAMMAR
/DEBUG 18
/LEXICAL
$END$ 1
EOLC 2
OR 3
RELOP 4
$($ 5
$)$ 6
NOT 7
AND 8
```

```
$CIRCUITDESCRIPTIONS$ 9
REMARK 10
CONSTANT 11
$,$ 12
$DEFINEDPARAMETERS$ 13
$=$ 14
$P$ 15
IAA 16
NUMBER 17
$GP$ 17
$W$ 19
IRA 20
KWG2A 21
KWG2 22
$K$ 23
$FCONV$ 24
$M$ 25
$-$ 26
$*$ 27
$R$ 28
$J$ 29
DERIV 30
ILA 31
$MODEL$ 32
$ELEMENTS$ 33
$E$ 34
$FREQ$ 36
KWTABLE 37
KWEQN 38
$FUNCTIONS$ 39
$D$ 40
KWG2B 41
$TIME$ 42
KWG4 43
KWG5 44
KWG6 45
KWG7C 47
KWG7D 48
$INITIALCONDITIONS$ 49
INTEGER 50
$MONTECARLO$ 51
AOP 52
$**$ 53
$MODELDESCRIPTIONS 55
$INITIAL$ 56
$PRINT$ 57
KWMODOP$ 58
$CHANGE$ 59
$SUPPRESS$ 60
$ALL$ 61
$PERM$ 62
$TEMP$ 63
MOP 64
```

```
$OPTIMIZATIONS$ 65
$OUTPUTS$ 66
$PLOTS$ 67
KWOUT 68
KWRC1 69
KWRC2 70
KWRC3 71
KWRC4 72
KWRC5 73
$IF$ 74
KWRC6 75
KWRC7 76
$RUNCONTROLS$ 77
$RERUNDESCRIPTIONS$ 78
KWICSH 79
KWRRC1 80
KWRRC2 81
KWRRC3 82
KWRRC4 83
KWRRC6 84
KWRRC7 85
$SENSITIVITY$ 86
KWDTABLE 87
KWVAR 88
$WORSTCASES$ 89
$FS 90
KWEXP 91
INTPAR 92
MFUNC 93
$/$ 94
$L$ 95
TYPE 96
/BNF
DECK=MODHD $END$ EOLC
DECK=CIRHD $END$ EOLC
DECK=MODHD CIRHD $END$ EOLC
DECK=MODHD CIRHD RERUNHD $END$ EOLC
DECK=CIRHD RERUNHD $END$ EOLC
BEXP=BTERM
BEXP=BEXP OR BTERM
BFACTOR=ME RELOP ME
BFACTOR=$($ BEXP $)$
BFACTOR=NOT BFACTOR
BTERM=BFACTOR
BTERM=BTERM AND BFACTOR
CHGLST=CHGLST1
CHGLST=CHGLST CHGLST1
CHGLST1=ELEMCH
CHGLST1=DEFPAR
CHGLST1=FUNCCH
CIRHD=$CIRCUITDESCRIPTIONS$ EOLC COMMENT
CIRHD=$CIRCUITDESCRIPTIONS$ EOLC
CIRHD=CIRHD CIRHD1
```

```
CIRHD1=ELEMSH
CIRHD1=DEFPARSH
CIRHD1=OUTSH
CIRHD1=ICSH
CIRHD1=FUNCSH
CIRHD1=RCSH
CIRHD1=SENSSH
CIRHD1=MCSH
CIRHD1=WCSH
CIRHD1=OPT
COMMENT=REMARK EOLC
COMMENT=COMMENT REMARK EOLC
CONLST=CONSTANT
CONLST=CONLST $,$ CONSTANT
DEFPAR=DEFPAR1
DEFPAR=DEFPAR2
DEFPAR=DEFPAR3
DEFPAR=DEFPAR4
DEFPARSH=$DEFINEDPARAMETERS$ EOLC
DEFPARSH=DEFPARSH DEFPAR EOLC
DEFPAR1=DP $=$ VALUE
DEFPAR2=$PS IAA $=$ NUMBER $($ NUM2 $)$
DEFPAR3=$GPS IAA $=$ GPLIST
DEFPAR4=$WS IAA $=$ DEFPAR40
DEFPAR40=DP
DEFPAR40=TABNAME
DEFPAR40=QNAME
DEFPAR40=XNAME
DEFPAR40=XFUNC
DEFPAR40=$($ ENTRY $,$ ENTRY $)$
DEFPAR40=TYPE
DP=$DPS IAA
DP=$PS IAA
DUMVLST=IRA
DUMVLST=DUMVLST $,$ IRA
DVAR=KWG2A KWG2 IAA
DVAR=DP
ELEM=$KS IAA $,$ NODELST $=$ $FCONVS ELEM3 $($ CONSTANT $)$
ELEM=G2 $,$ NODELST $=$ VALUE
ELEM=$MS IAA $,$ $LS IAA $-$ $LS IAA $=$ VALUE
ELEM=KWG2A IAA $,$ NODELST $=$ CONSTANT $*$ ELEM3 $RS IAA
ELEM=$JS IAA $,$ NODELST $=$ VALUE $*$ $JS IAA
ELEM=DERIV ELEM3 $=$ VALUE
ELEM=ILA $,$ NODELST $=$ $MODELS IAA
ELEM=ILA $,$ NODELST $=$ $MODELS IAA $($ MODOPS $)$
ELEM=ELEM4 IAA $,$ NODELST $=$ NUMBER $($ NUM2 $)$
ELEM=ELEM3 IAA $,$ NODELST $=$ $($ ENTRY $,$ ENTRY1
ELEMCH=$KS IAA $=$ $FCONVS ELEM3 $($ CONSTANT $)$
ELEMCH=G2 $=$ VALUE
ELEMCH=$MS IAA $=$ VALUE
ELEMCH=KWG2A IAA $=$ CONSTANT $*$ ELEM3 $RS IAA
ELEMCH=$JS IAA $=$ VALUE $*$ $JS IAA
```

```
ELEMCH=DERIV ELEM3 $=$ VALUE
ELEMCH=ELEM4 IAA $=$ NUMBER $($ NUM2 $)$
ELEMCH=ELEM5 IAA $=$ $($ ENTRY $,$ ENTRY1
ELEMSH=$ELEMENTS$ EOLC
ELEMSH=ELEMSH ELEM EOLC
ELEM3=$E$
ELEM3=$J$
ELEM4=ELEM3
ELEM4=$R$
ENTRY=CONSTANT
ENTRY=$FREQ$
ENTRY=KWTABLE IAA $($ G4 $)$
ENTRY=DP
ENTRY1=ENTRY $)$
ENTRY1=ENTRY $)$ $,$ TYPE
FUNC=KWEQN IAA $($ DUMVLST $)$ $=$ MEi EOLC
FUNCCH=KWTABLE IAA $=$ KWTABLE IAA
FUNCCH=KWEQN IAA $=$ KWEQN IAA
FUNCSH=$FUNCTIONS$ EOLC
FUNCSH=FUNCSH FUNC
FUNCSH=FUNCSH TABLE
GPLIST=GPLIST1
GPLIST=GPLIST GPLIST1
GPLIST1=$P$ IAA $*$ $D$ GPLIST2
GPLIST2=G5
GPLIST2=G6
G1=DP
G1=G2
G1=KWG2A G2
G1=KWG2B IAA
G1=$TIME$
G1=$FREQ$
G1=INTPAR
G2=KWG2 IAA
G3=G1
G3=TABNAME
G3=CONSTANT
G31ST=G3
G31ST=G31ST $,$ G3
G4=G2
G4=KWG4 IAA
G4=$TIME$
G4=$FREQ$
G4=INTPAR
G5=KWG5 IAA
G51ST=G5
G51ST=G51ST $,$ G5
G6=KWG6 IAA
G61ST=G6
G61ST=G61ST $,$ G6
G7=G2
G7=$M$ IAA
```

```
G7=DP
G7=KWG2A KWG2 IAA
G7=$W$ IAA
G7=$D$ KWG7C IAA
G7=INTPAR
G7=KWG7D $K$ IAA
G7=KWG2A KWG7D $K$ IAA
ICSH=$INITIALCONDITIONS$ EOLC
ICSH=ICSH KWICSH IAA $=$ NUMBER EOLC
INTLST=INTEGER
INTLST=INTLST $,$ INTEGER
MCSH=$MONTECARLO$ EOLC
MCSH=MCSH VARSET EOLC
ME=ADP MTERM
ME=MTERM
ME=ME ADP MTERM
MFACTOR=MFACTOR1
MFACTOR=MFACTOR1 $**$ MFACTOR1
MFACTOR1=$($ ME $)$
MFACTOR1=CONSTANT
MFACTOR1=IRA
MFACTOR1=MFUNC $($ VALUELIST $)$
MODENT=COMMENT
MODENT=ELEMSH
MODENT=DEFPARSH
MODENT=OUTSH
MODENT=FUNCSH
MODENT=ICSH
MODHD=$MODELDESCRIPTIONS$ EOLC
MODHD=$MODELDESCRIPTIONS$ $($ MODHD1 $)$ EOLC
MODHD=MODHD MODL
MODHD1=MODHD2
MODHD1=MODHD2 $,$ MODHD2
MODHD2=$INITIAL$
MODHD2=$PRINT$
MODL=$MODEL$ IAA $($ MODTYP $)$ $($ NODELST $)$ EOLC
MODL=$MODEL$ IAA $($ NODELST $)$ EOLC
MODL=MODL MODENT
MODOPS=MODOPS1
MODOPS=MODOPS $,$ MODOPS1
MODOPS1=KWMODOPS
MODOPS1=$CHANGE$ CHGLST
MODOPS1=$SUPPRESS$ OUTLST
MODOPS1=$SUPPRESS$ $ALL$
MODTYP=$PERM$
MODTYP=$TEMP$
MTERM=MFACTOR
```

```
MTERM=MTERM MOP MFACTOR
NODELST=IAA
NODELST=NODELST $-$ IAA
NUMLST=NUMLST1
NUMLST=NUMLST $,$ NUMLST1
NUMLST1=NUMBER
NUMLST1=NUMBER $($ NUMBER $,$ NUMBER $)$
NUM1=NUMBER
NUM1=NUM1 $,$ NUMBER
NUM2=NUMBER
NUM2=NUMBER $,$ NUMBER
OPT=$OPTIMIZATIONS EOLC
OPT=OPT VARSET EOLC
OUT=OUTLST
OUT=OUTLST OUT1
OUT=OUTLST OUT2
OUTLST=OUTLST1
OUTLST=OUTLST $,$ OUTLST1
OUTLST1=G7
OUTLST1=G7 $($ IAA $)$
OUTSH=$OUTPUTS$ EOLC
OUTSH=OUTSH OUT EOLC
OUT1=$,$ $PLOTS
OUT1=$,$ $PLOTS OUT3
OUT2=$,$ KWOUT
OUT2=$,$ KWOUT OUT4
OUT2=OUT4
OUT3=IAA
OUT3=IAA OUTLST1
OUT3=OUTLST1
OUT4=$,$ $PLOTS
OUT4=$,$ $PLOTS IAA
QNAME=KWEQN IAA $($ G$1ST $)$
RC=KWRC1 $=$ NUMBER
RC=KWRC2 $=$ KWRC3
RC=KWRC4
RC=KWRC5 $IF$ $($ BEXP $)$
RC=KWRC6 $=$ INTEGER
RC=KWRC7 $=$ CONSTANT
RCSH=$RUNCONTROLS$ EOLC
RCSH=RCSH RC EOLC
RERUNHD=$RERUNDESCRIPTIONS EOLC
RERUNHD=$RERUNDESCRIPTIONS $($ INTEGER $)$ EOLC
RERUNHD=RERUNHD RRUN
RRCIR=$ELEMENTS$ EOLC
RRCIR=RRCIR RRCIR1
RRCIR1=G2 $=$ NUMLST EOLC
RRDP=$DEFINEDPARAMETERS$ EOLC
RRDP=RRDP RRDP1
RRDP1=DP $=$ NUMLST EOLC
RRFUNC=$FUNCTIONS$ EOLC
RRFUNC=RRFUNC RRFUNC1
RRFUNC1=KWTABLE IAA $=$ NUM1 EOLC
```

```
RRFUNC1=RRFUNC1 NUM1 EOLC
RRIC=$INITIALCONDITIONS$ EOLC
RRIC=RRIC RRIC1 EOLC
RRIC1=KWICSH $=$ NUM1
RRRC=$RUNCONTROLS$ EOLC
RRRC=RRRC RRRC1 EOLC
RRRC1=KWRRC1 $=$ NUMLST
RRRC1=KWRRC2 $=$ KWRRC3
RRRC1=KWRRC4
RRRC1=KWRRC6 $=$ INILST
RRRC1=KWRRC7 $=$ CONLST
RRUN=COMMENT
RRUN=RRCIR
RRUN=RRDP
RRUN=RRIC
RRUN=RRFUNC
RRUN=RRRC
SENSSH=$SENSITIVITY$ EOLC
SENSSH=SENSSH VARSET EOLC
TABLE=KWDTABLE IAA EOLC
TABLE=TABLE NUMBER $,$ NUMBER EOLC
TABNAME=KWTABLE IAA
TABNAME=KWTABLE IAA $($ G1 $)$
VALUE=NUMBER
VALUE=TABNAME
VALUE=DP
VALUE=QNAME
VALUE=XNAME
VALUE=XFUNC
VALUELIST=ME
VALUELIST=VALUELIST $,$ ME
VAR=KWVAR IAA
VAR=DVAR
VAR=$FREQ$
VAR=INTPAR
VAR=$TIME$
VARLST=VARLST1
VARLST=VARLST $,$ VARLST1
VARLST1=VAR
VARLST1=CONSTANT
VARLST1=TABNAME
VARSET=$($ G51ST $/$ G61ST $)$
WCSH=$WORSTCASE$ EOLC
WCSH=WCSH VARSET EOLC
XFUNC=$F$ IAA $($ VARLST $)$
XNAME=KWEXP IAA $($ ME $)$
/END
**
```

APPENDIX I

CIRCUS-2 GRAMMAR

This appendix lists the grammar for the CIRCUS-2 language using BNF notation which is consistent with the input to the LR(k) Parser Generator. A listing of the undefined terminal symbols and their corresponding character strings is given below:

Lexical Symbol	Character String
ALPHASTRING	A string of alphameric characters
AOP	+ or -
APOSTROPHE	'
CHARG1	I, T, or *
CHOPTIONS	SHIFT or SCALE
COMPNAME	An alphameric identifier beginning with a letter with a maximum of four characters, representing a component name
DEVNAME	An alphameric identifier with a maximum of eight characters, representing a device name
DEVPAR	A device parameter name
ELPFX	R, C, L, K, V, J, PV, PJ, SV, SJ, TV, TJ, Z, or Y
EOLC	End of logical card symbol
FORTRAN	A block of FORTRAN code, including the END statement
FSPFX	V or J
IAA	An alphameric identifier
IAA2	An identifier containing only letters with a maximum of two characters, representing a model label prefix
ILA	An alphameric identifier beginning with a letter
INTEGER	An unsigned integer
INTERVAL	PRINT INTERVALS, PLOT INTERVALS, or RECORD INTERVALS
MFUNC	Any mathematical function name
MOP	* or /
RCLPFX	R, C, or L
STRING	Any character string
ZYPFX	Z or Y

The complete description of the CIRCUS-2 grammar is:

```
/COMMENT CIRCUS-2 GRAMMAR 7/22/75
/DEBUG 18
/COMMENT RCLPFX=R/L/C
/COMMENT FSPFX=V/J
/COMMENT ZYPFX=Z/Y
/COMMENT INTERVAL= PRINTINTERVALS/PLOTINTERVALS/RECORDINTERVALS
/COMMENT CHARGI= */I/I
/COMMENT CHOPTIONS=SHIFT/SCALE
/LEXICAL
$ENDOFJOB$ 20
EOLC 4
$END$ 21
ILA 5
$,$ 11
IAA 6
$($ 12
$)$ 13
$*$ 14
$=$ 15
$CHANGES$ 22
$GLOBALCHANGES$ 23
$DEVICENAMES$ 24
$MODELNAMES$ 25
$ARRAYS$ 26
$DEVICES$ 27
$ENDOFINPUTS$ 28
IAA2 7
$EQUATIONS$ 29
$RESTARTS$ 30
$EXECUTES$ 31
$EXTERNALNODES$ 32
FSPFX 33
$FUNCTIONS$ 34
$GLOBALDEVICEPARAMETERS$ 35
$DIAGNOSTICS$ 36
$HOLDFINALCONDITIONS$ 37
$SAVE$ 38
$STOPTIME$ 39
$INCREMENTSTOPTIME$ 40
$INITIALCONDITIONS$ 41
$LIMITS$ 42
$L$ 43
$MODELS$ 45
$DEFAULTDEVICEPARAMETERS$ 46
$FIXEDPARAMETERS$ 47
$D.C.STEADYSTATE$ 48
$MULTIPLEEXECUTE,$ 49
INTEGER 8
```

```
$ENDMULTIPLEEXECUTE$ 50
NUMBER 2
$OPTIMIZATIONCRITERION$ 51
$PRINT$ 52
$PLOT$ 53
$RECORD$ 54
$SINGLEVALUEDPARAMETER$ 55
$ONEDIMENSIONALTABLE$ 56
$TWODIMENSIONALTABLE$ 57
$PREFIX$ 59
ALPHASTRING 3
CHOPTIONS 59
RCLPFX 60
$REFERENCENODE$ 61
$CARDLENGTH$ 62
$LIST$ 63
$NOLIST$ 64
APOSTROPHE 9
STRING 10
$PLOTPAGELIMIT$ 65
$NOMODELLIBRARY$ 66
$NODEVICELIBRARY$ 67
$S$ 68
$STEADYSTATEGUESS$ 69
$STEADYSTATEPRINT$ 70
$T$ 71
$TOPOLOGY$ 72
ZYPFX 73
ELVOLT 74
FORTRAN 75
$P$ 76
ELCURR 77
ELNAME 78
DEVNAME 79
NODEVOLT 80
COMPNAME 81
DEVCLASS 82
OUTFUNC 83
INTERVAL 84
$O$ 85
DEVPAR 86
CHARG1 87
DEVPARAM 88
AOP 89
MOP 90
$**$ 91
MFUNC 92
ELPFX 93
/BNF
GOAL=DECK
ARGLIST=ILA
ARGLIST=ARGLIST $,$ ILA
ARRAYBLK=ARYBLK1 EOLC
```

```
ARRAYBLK=ARRAYBLK ARYBLK1 EOLC
ARYBLK1=IAA $,$ ARYVAL
ARYBLK1=ARYBLK1 $,$ ARYVAL
ARYVAL=VALUE
ARYVAL=$($VALUE $,$ VALUE $)$
CHARGS=CHARG1
CHARGS=CHARGS $,$ CHARG1
CHCIRC=$*$ IAA $=$ VALIST EOLC
CHCIRC=$*$ IAA $=$ VP EOLC
CHDEV=$CHANGE$ $,$ CHPAIRS EOLC
CHDEVPARAM=$CHANGE$ $,$ DEVICE $,$ VPLIST EOLC
CHGLOBAL=$GLOBALCHANGE$ $,$ VPLIST EOLC
CHNG=CHCIRC
CHNG=CHDEV
CHNG=CHDEVPARAM
CHNG=CHGLOBAL
CHPAIR1=$($OLDNAME$,$ DEVNAME $)$
CHPAIRS=CHPAIR1
CHPAIRS=CHPAIRS $,$ CHPAIR1
DECK=RUN $ENDOFJOB$ EOLC
DECK=RUN $END$ EOLC DECK
DEVBLK=$DEVICENAME$ $=$ IAA $,$ $MODELNAME$ $=$ IAA EOLC DEVBLK1
DEVBLK1=DEVBLK2
DEVBLK1=DEVBLK1 DEVBLK2
DEVBLK2=PARBLK
DEVBLK2=$ARRAYS$ EOLC ARRAYBLK
DEVCLASS=IAA2 $.$ IAA
DEVICE=DEVNAME
DEVICE=DEVCLASS
DEVICEBLK=$DEVICES$ EOLC DEVBLK $ENDOFINPUTS$ EOLC
DEVREF=IAA2 IAA $,$ NODELIST $,$ DEVNAME EOLC
DEVREF=IAA2 IAA $,$ NODELIST EOLC
ELCURR=$I$ $.$ ELNAME
ELEM=RCL
ELEM=K
ELEM=FIXEDSOURCE
ELEM=PSOURCE
ELEM=SINESOURCE
ELEM=TABSOURCE
ELEM=ZY
ELNAME=ELPFX IAA
ELNAME=ELNAME $.$ IAA
ELVOLT=$V$ $.$ ELNAME
EQNS=$EQUATIONS$ EOLC FORTRAN
EXEC1=EXEC1A
EXEC1=EXEC1 EXEC1A
EXEC1A=GLOBALDP
EXEC1A=FUNCLIST
EXEC1A=REFNODE
EXEC1A=DEVREF
EXEC1A=ELEM
EXEC1A=OUT
EXEC1A=MSS
EXEC1A=OPT
```

```
EXEC1A=$RESTART$ EOLC
EXEC1A=GRP1
EXEC1A=SSPRINT
EXEC2=EXEC4 EXEC3
EXEC2=EXEC2 EXEC4 EXEC3
EXEC3=MULTEX
EXEC3=$EXECUTE$ EOLC
EXEC4=EXEC5
EXEC4=EXEC4 EXEC5
EXEC5=GRP1
EXEC5=LIMIT
EXEC5=CHNG
EXTNODES=$EXTERNALNODES$ $=$ $($ NODELIST $)$ EOLC
FIXEDSOURCE=FSPFX IAA $,$ NODELIST $,$ VALUE EOLC
FUNC=IAA$($ARGLIST$)$ $=$ ME EOLC
FUNCLIST=$FUNCTIONS$ EOLC FUNCS $END$ EOLC
FUNCS=FUNC
FUNCS=FUNCS FUNC
GLOBALDP=$GLOBALDEVICEPARAMETERS$ EOLC PARBLOCK $ENDOFINPUT$ EOLC
GRP1=$DIAGNOSTICS$ EOLC
GRP1=$HOLDFINALCONDITIONS$ EOLC
GRP1=INCSTOP
GRP1=INITCOND
GRP1=OUTINTERVAL
GRP1=$SAVE$ EOLC
GRP1=$SGUESS
GRP1=$STOPTIME$ $=$ NUMBER EOLC
ICLIST=ICLISTA
ICLIST=ICLIST $,$ ICLISTA
ICLISTA=ELVOLT $=$ VALUE
ICLISTA=ELCURR $=$ VALUE
INCSTOP=$INCREMENTSTOPTIME$ $,$ VALUE EOLC
INITCOND=$INITIALCONDITIONS$ $,$ ICLIST EOLC
INITCOND=$INITIALCONDITIONS$ $=$ $0$ EOLC
K=$K$ IAA $,$ LNAME $,$ LNAME $,$ VALUE EOLC
LIMIT=$LIMITS$ $=$ INTEGER EOLC
LNAME=$L$ IAA
ME=AOP MTERM
ME=MTERM
ME=ME AOP MTERM
MFACTOR=MFACTOR1
MFACTOR=MFACTOR1 $**$ MFACTOR1
MFACTOR1=$($ ME $)$
MFACTOR1=NUMBER
MFACTOR1=ILA
MFACTOR1=MFUNC $($ VALUELIST $)$
MODBLK=MODEL
MODBLK=MODBLK MODEL
MODEL=$MODELNAME$ $=$ STRING EOLC MODSEC$ $ENDOFINPUT$ EOLC
MODELBLK=$MODEL$ EOLC MODBLK $ENDOFINPUT$ EOLC
MODSECA=EQNS
MODSECA =$DEFAULTDEVICEPARAMETERS$EOLC PARBLOCK
MODSECA=$FIXEDPARAMETERS$ $,$ ARGLIST EOLC
MODSECA=$ARRAYS$ $,$ ARGLIST EOLC
```

```
MODSECS=EXTNODES TOPO
MODSECS=MODSECS MODSECA
MSS=$D.C.STEADYSTATE$EOLC PARVARLIST $END$ EOLC
MTERM=MFACTOR
MTERM=MTERM MOP MFACTOR
MULTEX=$MULTIPLEEXECUTE,$ INTEGER EOLC MXBLOCK $ENDMULTIPLEEXECUTE$EOLC
MXBLK=CHNG
MXBLK=INCSTOP
MXBLOCK=MXBLK
MXBLOCK=MXBLOCK MXBLK
NODE=IAA
NODELIST=NODE
NODELIST=NODELIST $,$ NODE
NODEREF=IAA
NODEREF=NODEREF $,$ IAA
NUMLIST=NUMBER
NUMLIST=NUMLIST $,$ NUMBER
OLDNAME=DEVNAME
OLDNAME=COMPNAME
OPT=$OPTIMIZATIONCRITERION$ $=$ OUTVAR EOLC
OUT=$PRINT$ $,$ OUTLIST EOLC
OUT=$PLOT$ $,$ POUTLIST EOLC
OUT=$RECORD$ $,$ OUTLIST EOLC
OUTINTERVAL=INTERVAL $,$ NUMLIST EOLC
OUTLIST=OUTVAR
OUTLIST=OUTLIST $,$ OUTVAR
OUTPAIRS=VARPAIR
OUTPAIRS=OUTPAIRS $,$ VARPAIR
OUTVAR=ELVOLT
OUTVAR=ELCURR
OUTVAR=$VN$ $.$ NODEREF
OUTVAR=DEVPAR.COMPNAME
OUTVAR=MFUNC $($ VALUELIST $)$
PARBLK=PARBLKZ
PARBLK=PARBLK1
PARBLK=PARBLK2
PARBLKZ=$SINGLEVALUEDPARAMETERS$ EOLC PBLKZ EOLC
PARBLK1=$ONEDIMENSIONALTABLES$ EOLC PBLK1
PARBLK2=$TWODIMENSIONALTABLES$ EOLC PBLK2
PARBLOCK=PARBLK
PARBLOCK=PARBLOCK PARBLK
PARVARLIST=PVARA EOLC
PARVARLIST=PARVARLIST PVARA EOLC
PBLKZ=PBLKZA
PBLKZ=PBLKZ $,$ PBLKZA
PBLKZA=IAA $=$ VALUE
PBLK1=PBLK1A
PBLK1=PBLK1 PBLK1A
PBLK1A=IAA $,$ $($ VALIST $)$ $,$ $($ VALIST $)$ EOLC
PBLK2=PBLK2A
PBLK2=PBLK2 PBLK2A
PBLK2A=IAA $,$ $($ VALIST$),($VALIST$),($VALIST$)$ EOLC
POUTLIST=OUTLIST
```

```
POUTLIST=OUTLIST $,$ OUTPAIRS
PREFX=PREFX1
PREFX=PREFX PREFX1
PREFX1=$PREFIX$ $,$ IAA2 $=$ ALPHASTRING EOLC
PSOURCE=$PS FSPFX IAA $,$ NODELIST $,$ VALIST EOLC
PVARA=PVARB
PVARA=PVARA $,$ PVARB
PVARB=ELNAME $=$ $($ VALIST $)$
PVARB=DEVPAR $=$ $($ VALIST $)$
PVARB=DEVPAR $=$ CHOPTIONS $=$ $($VALIST$)$
RCL=RCLPFX IAA $,$ NODELIST $,$ VALUE EOLC
REFNODE=$REFERENCENODE$ $=$ NODE EOLC
RUN=RUNGRP
RUN=RUN RUNGRP
/COMMENT FOLLOWING RUNGRP PRODUCTIONS MAY OCCUR ARBITRARILY IN INPUT
RUNGRP=$CARDLENGTH$ $=$ INTEGER EOLC
RUNGRP=$LIST$ EOLC
RUNGRP=$NOLIST$ EOLC
/COMMENT FOLLOWING RUNGRP PRODUCTIONS MAY OCCUR ARBITRARILY, BUT NOT
/COMMENT INSIDE MODELBLK
RUNGRP=APOSTROPHE STRING EOLC
RUNGRP=$PLOTPAGELIMITS $=$ INTEGER EOLC
/COMMENT FOLLOWING RUNGRP PRODUCTIONS OCCUR IN GIVEN ORDER WHEN USED
RUNGRP=LIMIT
RUNGRP=$NOMODELLIBRARY$ EOLC
RUNGRP=$NODEVICELIBRARY$ EOLC
RUNGRP=MODELBLK
RUNGRP=DEVICEBLK
RUNGRP=PREFX
RUNGRP=EXEC1 EXEC3
RUNGRP=EXEC2
SINESOURCE=$S$ FSPFX IAA $,$ NODELIST $,$ VALIST EOLC
SSGUESS=$STEADYSTATEGUESS$ $,$ ICLIST EOLC
SSPRINT=$STEADYSTATEPRINT$ $,$ OUTLIST EOLC
TABSOURCE=$T$FSPFX IAA$,$NODELIST$,$ $($VALIST$)$ $,$ $($VALIST$)$EOLC
TOPO=$TOPOLOGY$ EOLC TOPOLIST
TOPOCARD=IAA $,$ NODE $,$ NODE EOLC
TOPOLIST=TOPOCARD
TOPOLIST=TOPOLIST TOPOCARD
VALFUNC=IAA $($ CHARGS $)$
VALIST=VALUE
VALIST=VALIST $,$ VALUE
VALUE=NUMBER
VALUELIST=ME
VALUELIST=VALUELIST $,$ ME
VARPAIR=$($ OUTVAR $,$ OUTVAR $)$
VP=VALFUNC
VP=VALUE
VP=CHOPTIONS $=$ VALFUNC
VP=$($ VALIST $),($ VALIST $)$
```

```
VPLIST=VPLISTA
VPLIST=VPLIST $,$ VPLISTA
VPLISTA=IAA $=$ VP
VPLISTA=IAA $=$ CHOPTIONS $=$ VALUE
ZY=ZYPFX IAA $,$ NODELIST $,$ VALUE $,$ $($ VALISTS),($ VALIST $)$ EOLC
/END

**
```

APPENDIX J

DESCRIPTION OF THE PARSER FOR THE LR(k) PARSER GENERATOR

This appendix lists the BNF input to the LR(k) Parser Generator which produces the LR(k) parsing tables for the LR(k) Parser Generator. The input listing constitutes a complete description of the grammar for the LR(k) Parser Generator, including identification of all terminal symbols and their lexical codes. Also included is specification of semantic routine linkages and error recovery. The intrinsic grammar is LR(1) and can be represented by 37 read states, 38 apply states, and 6 lookahead states.

A listing of the terminal symbols and the character strings which they represent is given below:

Lexical Symbol	Character String
COMMA	,
DYNAMIC	D.
EOLC	End of logical card symbol
EQ	=
I	Any identifier containing alphameric characters, starting with a letter
LP	(
NSEMANT	NSEMANT
RP	)
SBNF	BNF
SCOMMENT	COMMENT
SDEBUG	DEBUG
SEND	END
SLASH	/
SLEXICAL	LEXICAL
STAR	*
STRING	Any character string enclosed by $ symbols
TSTRING	Any character string enclosed by $ symbols

The lexical analyzer description for the LR(k) Parser Generator is given in Appendix K. The complete description of the parser for the LR(k) Parser Generator is:

```
/COMMENT $GRAMMAR FOR PARSER AND SDT GENERATORS
/LEXICAL 1
INTEGER 2
EOLC    4
SLASH 14
LP 15
RP 16
EQ 17
SLEXICAL 20
SBNF 21
SEND 22
SDEBUG 23
SCOMMENT 24
COMMA 25
NSEMANT 27
/LEXICAL 2
I 1
INTEGER 2
EOLC 4
SLASH 14
LP 15
STAR 18
COMMA 25
/LEXICAL 3
STRING 3
/LEXICAL 4
I 1
INTEGER 2
EOLC 4
SLASH 14
LP 15
COMMA 25
TSTRING 26
DYNAMIC 28
/BNF
S(7)=S1 BNF1 END1
S(7)=BNF1 END1
S1=S2
S1=S1 S2
S2(18)=(EOLC) SLASH NSEMANT INTEGER EOLC *
S2=LEX1
S2=COMM
LEX1(8)=(EOLC) SLASH SLEXICAL INTEGER EOLC *
LEX1=(EOLC) SLASH SLEXICAL EOLC *
```

```
LEX1=LEX1 LEX
BNF1=(EOLC) SLASH SBNF EOLC * BNFA
BNF1=BNF1 BNFA
BNFA=(EOLC) BNF2 EOLC *
BNFA=COMM
BNF2=BNFDEF
BNF2=BNF2 COMMA SDT
BNFDEF(5)=CLASS CODE EQ DEF
BNFDEF(5)=CLASS EQ DEF
CLASS(2)=I
CODE(3)=LP (RP) INTEGER RP *
DEF=TERM
DEF=DEF TERM
TERM(4)=I
TERM(10)=LP (RP) I RP *
TERM(21)=LP INTEGER RP
TERM(11)=STAR
LEX(6)=(EOLC) I INTEGER EOLC *
SDT=SDT1
SDT=SDT SDTTERM
SDT1=SDTTERM
SDT1=TRANS EQ SDTTERM
SDTTERM(19)=I
SDTTERM(12)=I LP (RP) INDEX RP *
SDTTERM(13)=TSTRING
SDTTERM(20)=DYNAMIC INTEGER
TRANS(14)=INTEGER
INDEX(15)=INTEGER
INDEX(16)=COMMA INTEGER
INDEX(17)=INTEGER COMMA INTEGER
COMM=SLASH SCOMMENT STRING
COMM=(EOLC) SLASH SDEBUG INTLIST EOLC *
INTLIST(9)=INTEGER
INTLIST(9)=INTLIST INTEGER
END1=(EOLC) SLASH SEND EOLC *
/END
**
```

APPENDIX K

DESCRIPTION OF THE LEXICAL ANALYZER FOR THE LR(k) PARSER GENERATOR

This appendix lists the input to the Lexical Analyzer Generator which produces the lexical analyzer tables for the LR(k) Parser Generator. The description of the lexical analyzer is:

```
/COMMENT LEXICAL ANALYZER FOR PARSER AND SDT GENERATOR
/MAXSCAN 4
/WORKSPACE 2
/HASHTABLE 751
/IDTABLE 2000
/CHARTABLE 49
/CHARCODE
A 1
B 2
C 3
D 4
E 5
F 6
G 7
H 8
I 9
J 10
K 11
L 12
M 13
N 14
O 15
P 16
Q 17
R 18
S 19
T 20
U 21
V 22
W 23
X 24
Y 25
Z 26
D0 27
D1 28
D2 29
D3 30
D4 31
D5 32
```

```
D6 33
D7 34
D8 35
D9 36
PLUS 37
MINUS 38
STAR 39
SLASH 40
LP 41
RP 42
DOLLAR 43
EQ 44
BLANK 45
COMMA 46
OUT 47
EOLC 48
NOCARD 49
/LEXCODE
INTEGER 2
STRING 3
EOLC 4
SLASH 14
LP 15
RP 16
EQ 17
STAR 18
LEXICAL 20
BNF 21
END 22
DEBUG 23
COMMENT 24
COMMA 25
TSTRING 26
NSEMANT 27
DYNAMIC 28
/SCANNER 1
/CLASS
3 B C D E I L N R S T
5 EOLC SLASH LP RP EQ COMMA
7 BLANK
8 D0 D1 D2 D3 D4 D5 D6 D7 D8 D9
9 A B C D E F G H I J K L M N O P Q R S T U V W X Y Z
/KEYWORDS
LEXICAL
BNF
END
DEBUG
COMMENT
NSEMANT
/FSA
$1,EOLC,EOLC,$2
$2,SLASH,SLASH,$3
$3,LP,LP,$4
```

```
$4,RP,RP,$5
$5,EQ,EQ,$6
$6,COMMA,COMMA,1
/SCANNER 2
/IDENTIFIER 1
/CLASS
4 A B C D E F G H I J K L M N O P Q R S T U V W X Y Z
5 SLASH EOLC COMMA
5 LP STAR
7 BLANK
8 DOLLAR
8 D0 D1 D2 D3 D4 D5 D6 D7 D8 D9
9 A B C D E F G H I J K L M N O P Q R S T U V W X Y Z
9 D0 D1 D2 D3 D4 D5 D6 D7 D8 D9
/FSA
$1,COMMA,COMMA,$2
$2,SLASH,SLASH,$3
$3,EOLC,EOLC,$4
$4,STAR,STAR,$5
$5,LP,LP,1
/SCANNER 3
/CLASS
5 A B C D E F G H I J K L M N O P Q R S T U V W X Y Z
5 D0 D1 D2 D3 D4 D5 D6 D7 D8 D9
5 PLUS MINUS STAR SLASH LP RP DOLLAR EQ BLANK COMMA DOT
/FSA
$1,5,$1,$2
$2,EOLC,STRING,1
/SCANNER 4
/IDENTIFIER 1
/CLASS
4 A B C E F G H I J K L M N O P Q R S T U V W X Y Z
5 COMMA LP EOLC D
7 BLANK
8 DOLLAR
8 D0 D1 D2 D3 D4 D5 D6 D7 D8 D9
9 A B C D E F G H I J K L M N O P Q R S T U V W X Y Z
9 D0 D1 D2 D3 D4 D5 D6 D7 D8 D9
/FSA
$1,COMMA,COMMA,$2
$2,EOLC,EOLC,$3
$3,LP,LP,$4
$4,D,$5,1
$5,DOT,DYNAMIC,4
/END
**
```

APPENDIX L

DESCRIPTION OF THE PARSER FOR THE LEXICAL ANALYZER GENERATOR

This appendix lists the BNF input to the LR(k) Parser Generator which produces the LR(k) parsing tables for the Lexical Analyzer Generator. The input listing constitutes a complete description of the grammar for the Lexical Analyzer Generator, including identification of all terminal symbols and their lexical codes. Also included is specification of semantic routine linkages and error recovery. The intrinsic grammar is LR(2) and can be represented by 60 read states, 47 apply states, and 8 lookahead states.

A listing of the terminal symbols and the character strings which they represent is given below:

Lexical Symbol	Character String
EOLC	End of logical card symbol
I	Any alphameric identifier, beginning with a letter
IDENT	IDENTIFIER
INTEGER	An unsigned integer
STRING	Any character string enclosed by $ symbols

The lexical analyzer description for the Lexical Analyzer Generator is given in Appendix M. The complete description of the parser for the Lexical Analyzer Generator is:

```
/COMMENT $GRAMMAR FOR PARSER FOR LEXICAL ANALYZER GENERATORS
/LEXICAL 1
INTEGER 2
EOLC 4
$/$ 14
$,$ 15
$*$ 16
$$$$ 17
$WORKSPACES 19
$HASHTABLES 20
$IDTABLES 21
$ENDS 22
$CHARTABLES 23
$CHARCODES 24
```

```
$LEXCODE$ 25
$SCANNER$ 26
$CLASS$ 28
$KEYWORDS$ 29
$KWLENGTHS$ 30
$FSA$ 31
$DEBUG$ 32
$MAXSCAN$ 33
$COMMENT$ 34
IDENT 36
$L$ 37
/LEXICAL 2
I 1
INTEGER 2
EOLC 4
$/$ 14
$,$ 15
$*$ 16
$$$$ 17
/LEXICAL 3
STRING 35
/BNF
S(26)=GLOBAL1 GLOBAL2 SCAN END1
GLOBAL1(10)=GLOBAL
GLOBAL=GLB
GLOBAL=GLOBAL GLB
GLB(2)=(EOLC) $/$ $WORKSPACE$ INTEGER EOLC *
GLB(3)=(EOLC)$/$ $HASHTABLE$ INTEGER EOLC *
GLB(4)=(EOLC) $/$ $IDTABLE$ INTEGER EOLC *
GLB(5)=(EOLC) $/$ $CHARTABLE$ INTEGER EOLC *
GLB=(EOLC) $/$ $CHARCODE$ EOLC * CHRCOD
GLB(24)=(EOLC) $/$ $MAXSCAN$ INTEGER EOLC *
GLB=COMM
GLOBAL2=(EOLC) $/$ $LEXCODE$ EOLC * LEX
CHRCOD(6)=(EOLC) I INTEGER EOLC*
CHRCOD(6)=(EOLC) CHRCOD I INTEGER EOLC*
LEX=LEX1
LEX=LEX LEX1
LEX1(7)=(EOLC) I INTEGER EOLC *
LEX1(13)=(EOLC) I EOLC *
LEX1(9)=(EOLC) I INTEGER LEX2 EOLC *
LEX2(28)=INTEGER
LEX2(29)=LEX2 $L$ INTEGER INTEGER
SCAN=SCAN1
SCAN=SCAN SCAN1
SCAN1(8)=(EOLC) $/$ $SCANNER$ INTEGER EOLC *
SCAN1=SCAN1 SCAN2
SCAN2=SCANCLS
SCAN2=SCANKW
SCAN2(15)=SCANKWL
SCAN2(25)=SCANFSA
```

```
SCAN2=COMM
SCAN2(27)=(EOLC) $/$ IDENT INTEGER EOLC *
SCANCLS=(EOLC) $/$ $CLASS$ EOLC * CLASS1
SCANCLS=SCANCLS CLASS1
CLASS1=(EOLC) CLASSNO IDLIST EOLC *
CLASSNO(11)=INTEGER
IDLIST(12)=I
IDLIST(12)=IDLIST I
SCANKW=(EOLC) $/$ $KEYWORDS$ EOLC * LEX
SCANKWL=(EOLC) $/$ $KWLENGTHS$ INTLIST EOLC *
INTLIST(14)=INTEGER
INTLIST(14)=INTLIST INTEGER
SCANFSA=(EOLC) $/$ $FSAS$ EOLC * FSA1
SCANFSA=SCANFSA FSA1
FSA1(22)=(EOLC)$$$$ INTEGER $,$ SYMBOL $,$ DEST $,$ DEST EOLC *
SYMBOL(16)=I
SYMBOL(17)=INTEGER
DEST(18)=$$$$ INTEGER
DEST(19)=I
DEST(20)=INTEGER
DEST(21)=$*$ INTEGER
COMM=$/$ $COMMENTS$ STRING
COMM=(EOLC) $/$ $DEBUG$ DLIST EOLC *
DLIST(23)=INTEGER
DLIST(23)=DLIST INTEGER
END1=(EOLC) $/$ $END$ EOLC *
/END
**
```

APPENDIX M

DESCRIPTION OF THE LEXICAL ANALYZER FOR THE LEXICAL ANALYZER GENERATOR

This appendix lists the input to the Lexical Analyzer Generator which produces the lexical analyzer tables for the Lexical Analyzer Generator. The description of the lexical analyzer is:

```
/COMMENT DESCRIPTION OF LEXICAL ANALYZER FOR LEXICAL ANALYZER GENERATOR
/MAXSCAN 5
/WORKSPACE 2
/HASHTABLE 751
/IDTABLE 2000
/CHARTABLE 49
/CHARCODE
A 1
B 2
C 3
D 4
E 5
F 6
G 7
H 8
I 9
J 10
K 11
L 12
M 13
N 14
O 15
P 16
Q 17
R 18
S 19
T 20
U 21
V 22
W 23
X 24
Y 25
Z 26
D0 27
D1 28
D2 29
D3 30
D4 31
D5 32
D6 33
D7 34
```

```
D8 35
D9 36
PLUS 37
MINUS 38
STAR 39
SLASH 40
LP 41
RP 42
DOLLAR 43
EQ 44
BLANK 45
COMMA 46
DOT 47
EOLC 48
NOCARD 49
/LEXCODE
SLASH 14
WORKSPACE 19
INTEGER 2
EOLC 4
HASHTABLE 20
IDTABLE 21
CHARTABLE 23
CHARCODE 24
LEXCODE 25
SCANNER 26
CLASS 28
KEYWORDS 29
KWLENGTH 30
FSA 31
DOLLAR 17
COMMA 15
STAR 16
END 22
DEBUG 32
MAXSCAN 33
COMMENT 34
STRING 35
IDENTIFIER 36
L 37
/SCANNER 1
/CLASS
1 A B G J O P Q R U V X Y Z PLUS MINUS LP RP EQ DOT
3 C D E F H I K L M N S T W
5 SLASH EOLC DOLLAR COMMA STAR
7 BLANK
8 D0 D1 D2 D3 D4 D5 D6 D7 D8 D9
9 A B C D E F G H I J K L M N O P Q R S T U V W X Y Z
/KEYWORDS
WORKSPACE
HASHTABLE
IDTABLE
CHARTABLE
```

```
CHARCODE
LEXCODE
SCANNER
CLASS
KEYWORDS
KWLENGTH
FSA
END
DEBUG
MAXSCAN
COMMENT
IDENTIFIER
L
/FSA
$1,SLASH,SLASH,$2
$2,EOLC,EOLC,$3
$3,DOLLAR,DOLLAR,$4
$4,COMMA,COMMA,$5
$5,STAR,STAR,1
/SCANNER 2
/IDENTIFIER 1
/CLASS
4 A B C D E F G H I J K L M N O P Q R S T U V W X Y Z
4 PLUS MINUS LP RP EQ DOT
5 SLASH EOLC DOLLAR COMMA STAR
7 BLANK
8 D0 D1 D2 D3 D4 D5 D6 D7 D8 D9
9 A B C D E F G H I J K L M N O P Q R S T U V W X Y Z
9 D0 D1 D2 D3 D4 D5 D6 D7 D8 D9
9 PLUS MINUS LP RP EQ DOT
9 SLASH DOLLAR STAR
/FSA
$1,SLASH,SLASH,$2
$2,EOLC,EOLC,$3
$3,DOLLAR,DOLLAR,$4
$4,COMMA,COMMA,$5
$5,STAR,STAR,1
/SCANNER 3
/CLASS
5 A B C D E F G H I J K L M N O P Q R S T U V W X Y Z
5 D0 D1 D2 D3 D4 D5 D6 D7 D8 D9
5 PLUS MINUS STAR SLASH LP RP DOLLAR EQ BLANK COMMA DOT
/FSA
$1,5,$1,$2
$2,EOLC,STRING,1
/END
**
```

APPENDIX N

EXECUTIVE CONTROL LANGUAGE GRAMMAR

This appendix lists the grammar for the Executive Control Language using BNF notation which is consistent with the input to the LR(k) Parser Generator. The grammar is LR(1) and the parser generated by the grammar has 71 read states, 55 apply states, and 11 lookahead states.

A listing of the terminal symbols and the character strings which they represent is given below:

Lexical Symbol	Character String
AND	.AND.
AOP	+ or -
CCARD	An arbitrary control card image
CHRSTRNG	Any arbitrary character string
COMMENT	An arbitrary comment, ending with EOLC
EOLC	End of logical card symbol
ILA	Alphameric identifier, starting with a letter
LABEL	An unsigned nonzero integer of one to five digits
MOP	* or /
NOT	.NOT.
NUMBER	A real number representation
OR	.OR.
PROCNAME	A procedure name
RELOP	.EQ., .NE., .LE., .LT., .GE., or .GT.

The grammar for ECL is:

```
/COMMENT GRAMMAR FOR EXECUTIVE CONTROL LANGUAGE
/LEXICAL
AND 1
AOP 2
$CALL$ 3
$CC$ 4
CCARD 5
CHRSTRNG 6
COMMENT 7
$CONTINUE$ 8
```

```
$DEFINE$ 9
$DELETE$ 10
$END$ 11
$ENDCC$ 12
EOLC 13
$GOTO$ 14
$IF$ 15
ILA 16
LABEL 17
$LIST$ 18
MOP 19
NOT 20
NUMBER 21
OR 22
PROCNAME 23
$READ$ 24
RELOP 25
$RETURN$ 26
$STORE$ 27
$WRITE$ 28
$,$ 29
$($ 30
$)$ 31
$/$ 32
$**$ 33
$$$$ 34
$=$ 35
/BNF
ECL=ECLCARDS ECLEND
ECL=PROCSEC PROCEND ECLCARDS ECLEND
ECL=PROCSEC PROCEND ECLEND
ARG=STRING
ARG=ME
ARG1=ILA
ARG1=NUMBER
ARGLIST=ARGLIST $,$ ARG
ARGLIST=ARG
BEXP=BEXP OR BTERM
BEXP=BTERM
BFACTOR=NOT BFACTOR
BFACTOR=$($ BEXP $)$
BFACTOR=ME RELOP ME
BTERM=BTERM AND BFACTOR
BTERM=BFACTOR
CCLIST=CCARD EOLC
CCLIST=CCLIST CCARD EOLC
CCSECT=$/$ $CC$ $/$ EOLC CCLIST $/$ $ENDCC$ $/$ EOLC
ECLCARDS=ECLCARDS ENTRY
ECLCARDS=ENTRY
ECLEND=$END$ EOLC
```

```
ENTRY=LABEL IFCLAUSE STATEMENT EOLC
ENTRY=LABEL STATEMENT EOLC
ENTRY=IFCLAUSE STATEMENT EOLC
ENTRY=STATEMENT EOLC
ENTRY=CCSECT
ENTRY=COMMENT
IFCLAUSE=$IF$ $($ BEXP $)$
ME=AOP MTERM
ME=MTERM
ME=ME AOP MTERM
MFACTOR=MFACTOR1
MFACTOR=MFACTOR1 $**$ MFACTOR1
MFACTOR1=ARG1
MFACTOR1=$($ ME $)$
MTERM=MTERM MOP MFACTOR
MTERM=MFACTOR
PARAM=ILA
PARAM=$$$$ ILA $$$$
PARLST=PARLST $,$ PARAM
PARLST=PARAM
PROCCOM=$LIST$ EOLC
PROCCOM=$DEFINE$ $,$ PROCDEF
PROCCOM=$STORE$ $,$ PROCDEF
PROCCOM=$DELETE$ $,$ PROCNAME EOLC
PROCDEF=PROCHDR EOLC ECLCARDS ECLEND
PROCEND=$END$ PROCNAME EOLC
PROCHDR=PROCNAME $($ PARLST $)$
PROCHDR=PROCNAME
PROCREF=PROCNAME
PROCREF=PROCNAME $($ ARGLIST $)$
PROCSEC=PROCSEC PROCCOM
PROCSEC=PROCCOM
REPLACE=ILA $=$ ME
REPLACE=ILA $=$ STRING
STATEMENT=$CONTINUE$
STATEMENT=$GOTO$ LABEL
STATEMENT=$RETURN$
STATEMENT=REPLACE
STATEMENT=$CALL$ PROCREF
STATEMENT=$WRITE$ $($ VALLIST $)$
STATEMENT=$READ$
STRING=$$$$ CHRSTRNG $$$$
VALLIST=ILA
VALLIST=VALLIST $,$ ILA
/END
**
```

APPENDIX O

STRUCTURED PROGRAMMING CONVENTIONS

The programming of the software described in this report has been accomplished using structured programming techniques. This has been done in spite of the fact that the programs are written in FORTRAN IV, perhaps one of the most unsuitable languages in general use with which to use structured programming. The structure conventions are explained below. These conventions are used in both the actual software itself and in algorithms which are given in this report. The use of this structuring method eliminates the need for all flow charts and provides a self documenting capability for the software itself.

Four types of control structures are used; namely, sequential, DO WHILE, IF-THEN-ELSE, and CASE control structures. By suitable combination of these four types programs and algorithms of any complexity may be written. These basic structures are combined with an indentation notation which is used to delineate the bounds of each structure in the program or algorithm.

Since FORTRAN does not permit the keywords DO WHILE, IF-THEN-ELSE, and CASE to be used, these keywords are written as structure comments. This permits the predicates for DO WHILE, IF-THEN, and CASE to be written in more natural terms since they do not have to be intelligible to the compiler. They are followed immediately in the code by the FORTRAN implementation of the structure comment. Thus, the programmer has the ability to write programs in true structured style using language elements of his own choosing, along with the implementation of the structure using the FORTRAN language. Thus, coding and documentation are carried along in one-to-one correspondence.

In addition, all algorithms and programs were developed in a topdown fashion using successive redefinition. Program implementation in code was not begun until entire major sections had been completely written in structured comment form. Experience with this technique has shown that several programs can be simultaneously implemented, errors are greatly

reduced, and program complexity can be extended by an order of magnitude with no loss of intellectual manageability compared to programs written using former program development techniques.

In the structured comment format, each comment is begun with an asterisk to visually delineate comment lines from coding lines. A comment may be continued on the next line; however, the next line does not have the asterisk header and starts one column farther in than the first line. Thus, if, for a given indentation level, the comment asterisk appears in column n, the first character of the comment continuation will start in column n + 1. All FORTRAN statements also start at column n + 1; if a FORTRAN statement is continued, all continuation cards start at column n + 2. Thus, the structured skeleton is easily followed by the identifying asterisks which introduce the structured comments. This same convention will be used in specifying algorithms in this report, except that they will not include any FORTRAN coding. Each subsequent indentation level is started three columns farther to the right from the previous level to indicate nested structures.

Sequential blocks are written on the same indentation level as in this example:

```
*INITIALIZE CONTROL VARIABLES
*READ INPUT DATA
*BUILD DATA STRUCTURE
*TRAVERSE DATA STRUCTURE, GENERATING OUTPUT
```

The IF-THEN-ELSE structure is illustrated by:

```
*IF(predicate)THEN
   *BLOCK 1
*ELSE
   *BLOCK 2
*BLOCK 3
```

If the predicate is satisfied, then BLOCK 1 is executed. BLOCK 1 is automatically terminated by the appearance of the ELSE on the same level as the IF-THEN. Thus, following the completion of BLOCK 1, control is passed to BLOCK 3 (which is actually a sequential block following the IF-THEN-ELSE block). If the predicate is not satisfied, then BLOCK 2 is executed, followed by BLOCK 3. An alternate form of the IF-THEN-ELSE structure occurs when there is no ELSE part:

```
*IF(predicate)THEN
   *BLOCK 1
*BLOCK 3
```

If the predicate is satisfied, then BLOCK 1 is executed, followed by BLOCK 3. If the predicate is not satisfied, BLOCK 1 is not executed, and control passes immediately to the execution of BLOCK 3.

The DO WHILE structure has the form:

```
*DO WHILE(predicate)
   *BLOCK 1
*BLOCK 2
```

Here the predicate is tested immediately upon entry to the structure. If the predicate is satisfied, then BLOCK 1 is executed, followed by a return to the predicate test. BLOCK 1 is repetitively executed until the predicate is no longer satisfied, at which time control passes to BLOCK 2. Note that if the predicate is not satisfied the first time, then BLOCK 1 is not executed at all. A variant of the DO WHILE is the DO, sometimes a more natural means of expressing the conditions for executing the subordinate block. Its action is identical to the DO WHILE:

```
*DO(predicate)
   *BLOCK 1
*BLOCK 2
```

The last type of control structure is the CASE structure. This is used to select one of several possible blocks, depending upon the value of an expression. Its structure can be seen by the following example:

```
*CASE(expression)
   *Expression value = 1
      *BLOCK 1
   *Expression value = 2
      *BLOCK 2
   .
   .
   .
   *Expression value = n
      *BLOCK n
*BLOCK A
```

The expression is evaluated and control is passed to a particular block, depending upon the expression value. Following completion of the specified block, control is passed out of the CASE structure to the next sequential block, BLOCK A.

Combining these ideas permits us to write algorithms of any complexity. As an example, consider the following algorithm for traversing an n-ary tree, where each node of the tree is represented by a linked list of sibling cells, each of which has a pointer to a descendant node. Thus, each sibling cell has two pointers, one for the next sibling cell and one for the descendant node.

TRAVERSE N-ARY TREE ALGORITHM

```
*ENTER ROOT NODE
*TRAVERSE=TRUE
*BACKUP=FALSE
*DO WHILE(TRAVERSE)
   *IF(NODE CELL HAS DESCENDANT AND BACKUP=FALSE)THEN
      *DESCEND TO DESCENDANT CELL
   *ELSE
      *IF(NODE CELL HAS SIBLING CELL)THEN
         *MOVE TO SIBLING CELL
         *BACKUP=FALSE
      *ELSE
         *BACKUP TO PARENT CELL
         *IF(ROOT NODE)THEN
            *TRAVERSE=FALSE
         *BACKUP=TRUE
*STOP
```

Frequently, the STOP step is implied; that is, when the algorithm drops off the end it is considered to be finished.

BIBLIOGRAPHY

Aho, A. V. and Ullman, J. D., The Theory of Parsing, Translation, and Compiling: Vol. 1 Parsing, Vol. 2 Compiling, Prentice-Hall, Inc.: Englewood Cliffs, N. J., 1972.

Altman, V. E., "A Language Implementation System," AD 780672, Massachusetts Institute of Technology, Cambridge, Mass., May 1974.

Becker, David, "Extended SCEPTRE, User's Manual," AFWL-TR-73-75, Vol. 1, Air Force Weapons Laboratory, Kirtland Air Force Base, New Mexico, Dec. 1974.

Dahl, O.-J., Dijkstra, E. W., and Hoare, C. A. R., Structured Programming, Academic Press, London, England, 1972.

Dembart, B., and Milliman, L., "CIRCUS-2, A Digital Computer Program For Transient Analysis of Electronic Circuits, Volume 1, User's Guide," HDL-0062-1, Harry Diamond Laboratories, Washington, D. C., June 1973.

De Remer, F. L., "Simple LR(k) Grammars," CACM 14, 7, July 1971, pp. 453-460.

Glatt, C. R., et al., "DIALOG: An Executive Computer Program for Linking Independent Programs, Final Report," N73-32090, Aerophysics Research Corp., Hampton, Va., 1973.

Gries, D., Compiler Construction for Digital Computers, John Wiley and Sons, Inc., New York, 1971.

Hopcroft, J. E. and Ullman, J. D., Formal Languages and their Relation to Automata, Addison-Wesley, Inc., Reading, Mass., 1969.

James, L. R., "A Syntax Directed Error Recovery Method," M. S. Thesis, Univ. of Toronto, Toronto, Canada, 1971.

Knuth, D. E., "On the Translation of Languages from Left to Right," Inf. Control 8, Oct. 1965, pp. 607-639.

Korenjak, A., "A Practical Method for Constructing LR(k) Processors," CACM 12, 11, Nov. 1969, pp. 613-623.

Lalonde, W. R., "An Efficient LALR Parser Generator," Tech. Report. CSRG-2, Univ. of Toronto, Toronto, Canada, 1971.

Leinius, R. P., "Error Detection and Recovery for Syntax Directed Compiler Systems," Ph.D. Thesis, Univ. of Wisconsin, Madison, Wisconsin, 1970.

Malmberg, A. F., "NET-2 Network Analysis Program, Release 9, User's Manual," HDL-050-1, Harry Diamond Laboratories, Washington, D. C., Sept. 1973.

DISTRIBUTION LIST

DEPARTMENT OF DEFENSE

Director
Armed Forces Radiobiology Research Institute
ATTN: Robert E. Carter
ATTN: Tech. Lib.

Director
Defense Communications Agency
ATTN: Code 540.5
ATTN: Code 930, Monte I. Burgett, Jr.

Defense Documentation Center
12 cy ATTN: TC

Director
Defense Nuclear Agency
ATTN: DDST
ATTN: STSI
3 cy ATTN: STTL, Tech. Lib.

Headquarters
European Command
J-5
ATTN: ECJ6-PF

AUL
ATTN: LDE

Commander
Field Command
Defense Nuclear Agency
ATTN: FCPR

Chief
Livermore Division, Field Command, DNA
ATTN: FCPRL

OJCS/J-3
ATTN: J-3, RDTA Br., WWMCCS, Plans Div.

DEPARTMENT OF THE ARMY

Commander
Frankford Arsenal
ATTN: SARFA-FCD, Marvin Elnick

Commander
Harry Diamond Laboratories
ATTN: AMXDO-EM, R. Bostak
ATTN: AMXDO-RB, Joseph R. Miletta
ATTN: AMXDO-RBI, John A. Rosado
ATTN: AMXDO-RCC, John E. Thompkins
ATTN: AMXDO-EM, Robert F. Gray
ATTN: AMXDO-EM, Raphael Wong
ATTN: AMXDO-RC, Robert B. Oswald, Jr.
ATTN: AMXDO-EM, J. W. Beilfuss

Commander
Picatinny Arsenal
ATTN: SMUPA-FR-S-P, Lester W. Doremus
ATTN: SARPA-ND-N
ATTN: SARPA-ND-C-E, Amina Nordio

Commander
TRASANA
ATTN: ATAA-EAC, Francis N. Winans

DEPARTMENT OF THE ARMY (Continued)

Director
U.S. Army Ballistic Research Labs.
ATTN: AMXBR-X, Julius J. Meszaros
ATTN: AMXBR-VL, John W. Kinch
ATTN: AMXRD-BVL, David L. Rigotti

Commander
U.S. Army Electronics Command
ATTN: AMSEL-GG-TD, W. R. Werk
ATTN: AMSEL-TL-MD, Gerhart K. Gaule
ATTN: AMSEL-TL-IR, Edwin T. Hunter

Commanding Officer
U.S. Army Electronics Command
Night Vision Laboratory
ATTN: CPT Allan S. Parker

Commander
U.S. Army Electronics Proving Ground
ATTN: STEEP-MT-M, Gerald W. Durbin

Commandant
U.S. Army Field Artillery School
ATTN: ATSFA-CTD-ME, Harley Moberg

Commander
U.S. Army Mat. & Mechanics Rsch. Ctr.
ATTN: AMXMR-HH, John F. Dignam

Commander
U.S. Army Materiel Dev. & Readiness Cmd.
ATTN: AMCRD-WN-RE, John F. Corrigan

Commander
U.S. Army Missile Command
ATTN: AMSI-RGP, Victor W. Ruwe
ATTN: AMSMI-RGP, Hugh Green
ATTN: AMSMI-RRR, Faison P. Gibson

Commander
U.S. Army Nuclear Agency
ATTN: ATCN-W, LTC Leonard A. Sluga

Project Manager
U.S. Army Tactical Data Systems, AMC
ATTN: Dwaine B. Huewe

Commander
White Sands Missile Range
ATTN: STEWS-TE-NT, Marvin P. Squires

DEPARTMENT OF THE NAVY

Chief of Naval Research
Navy Department
ATTN: Code 427

Commanding Officer
Naval Ammunition Depot
ATTN: Code 7024, James Ramsey

Commander
Naval Electronics Laboratory Center
ATTN: H. F. Wong

DEPARTMENT OF THE NAVY (Continued)

Commander
Naval Electronic Systems Command
ATTN: ELEX 05323, Cleveland F. Watkins
ATTN: PME 117-21
ATTN: Code 50451

Director
Naval Research Laboratory
ATTN: Code 2627, Doris R. Folen

Commander
Naval Surface Weapons Center
ATTN: Code 431, Edwin B. Dean
ATTN: Code WX-21, Tech. Lib.

Commander
Naval Surface Weapons Center
Dahlgren Laboratory
ATTN: Code FUR, Robert A. Amadori

Commanding Officer
Naval Weapons Evaluation Facility
ATTN: Code ATG, Mr. Stanley

Director
Strategic Systems Project Office
ATTN: SP-2701, John W. Pitsenberger

DEPARTMENT OF THE AIR FORCE

Commander
Aeronautical Systems Division, AFSC
ATTN: ASD-YH-EX, Lt Col Robert Leverette

AF Aero-Propulsion Laboratory, AFSC
ATTN: POD, P. E. Stover

AF Weapons Laboratory, AFSC
ATTN: ELA
ATTN: ELC
ATTN: ELP, Carl E. Baum
ATTN: HO
2 cy ATTN: SUL

AFTAC
ATTN: TAE

Air Force Avionics Laboratory, AFSC
ATTN: AFAL, TEA, Hans J. Hennecke
ATTN: AFAL, AAA

Commander
Foreign Technology Division, AFSC
ATTN: FTD, PDJC

Commander
Ogden Air Logistics Center
ATTN: MMEWM, Robert Joffs

SAMSO/DY
ATTN: DYS, Maj Larry A. Darda

SAMSO/YD
ATTN: YDD, Maj Marion P. Schneider

Commander in Chief
Strategic Air Command
ATTN: XPFS, Maj Brian G. Stephan

ENERGY RESEARCH & DEVELOPMENT ADMINISTRATION

University of California
Lawrence Livermore Laboratory
ATTN: Donald J. Meeker, L-153
ATTN: E. K. Miller, L-156
ATTN: Lawrence Cleland, L-156
ATTN: Frederick R. Kovar, L-94
ATTN: William J. Hogan, L-531

Los Alamos Scientific Laboratory
ATTN: Doc. Con. for Bruce W. Noel
ATTN: Doc. Con. for J. Arthur Freed

DEPARTMENT OF DEFENSE CONTRACTORS

Aerojet Electro-Systems Co., Div.
Aerojet-General Corporation
ATTN: Thomas D. Hanscome

Aeronutronic Ford Corporation
Aerospace & Communications Ops.
ATTN: E. R. Poncelet, Jr.
ATTN: Ken C. Attinger
ATTN: Tech. Info. Section

Aeronutronic Ford Corporation
ATTN: Samuel R. Crawford, M.S. 531

Avco Research & Systems Group
ATTN: Research Library, A-830, Rm. 7201

The BDM Corporation
ATTN: T. H. Neighbors
ATTN: William Druen

The BDM Corporation
ATTN: Allan F. Malmberg

Bell Aerospace Company
Division of Textron, Inc.
ATTN: Carl B. Schoch, Wpns. Effects Grp.

The Bendix Corporation
Communication Division
ATTN: Doc. Con.

The Bendix Corporation
Research Laboratories Division
ATTN: Mgr., Prgm. Dev., Donald J. Niehaus

The Boeing Company
ATTN: Robert S. Caldwell, M.S. 2R-00
ATTN: Donald W. Egelkrout, M.S. 2R-00
ATTN: David L. Dye, M.S. 87-75
ATTN: Aerospace Library
ATTN: Howard W. Wicklein, M.S. 17-11

Booz-Allen & Hamilton, Inc.
ATTN: Raymond J. Chrisner

Brown Engineering Company, Inc.
ATTN: David L. Lambert, M.S. 18

Charles Stark Draper Laboratory, Inc.
ATTN: Kenneth Fertig
ATTN: Paul R. Kelly

Computer Sciences Corporation
ATTN: Richard H. Dickhaut

DEPARTMENT OF DEFENSE CONTRACTORS (Continued)

Cutler-Hammer, Inc.
AIL Division
ATTN: Anne Anthony, Central Tech. Files

University of Denver
Colorado Seminary
ATTN: Sec. Officer for Fred P. Venditti

The Dikewood Corporation
ATTN: L. Wayne Davis

E-Systems, Inc.
Greenville Division
ATTN: Library, 8-50100

Exp. & Math. Physics Consultants
ATTN: Thomas M. Jordan

Fairchild Industries, Inc.
ATTN: Mgr., Config. Data & Standards

The Franklin Institute
ATTN: Ramie H. Thompson

Garrett Corporation
ATTN: Robert E. Weir, Dept. 93-9

General Electric Company
Space Division
ATTN: Larry I. Chasen
ATTN: John R. Greenbaum
ATTN: Joseph C. Peden, CCF 8301
ATTN: John L. Andrews
ATTN: James P. Spratt

General Electric Company
Re-Entry & Environmental Systems Div.
ATTN: Robert V. Benedict

General Electric Company
TEMPO-Center for Advanced Studies
ATTN: DASIAC
ATTN: William McNamera
ATTN: Royden R. Rutherford
ATTN: M. Espig

General Electric Company
ATTN: CSP 0-7, L. H. Dee

General Electric Company
Aerospace Electronics Systems
ATTN: W. J. Patterson, Drop 233
ATTN: George Francis, Drop 233

General Electric Company-TEMPO
ATTN: DASIAC for William Alfonte

General Research Corporation
ATTN: Robert D. Hill

General Research Corporation
Washington Operations
ATTN: David K. Osias

GTE Sylvania, Inc.
Electronics Systems Grp.-Eastern Div.
ATTN: Leonard L. Blaisdell
ATTN: James A. Waldon

DEPARTMENT OF DEFENSE CONTRACTORS (Continued)

GTE Sylvania, Inc.
ATTN: Charles H. Ramsbottom
ATTN: Herbert A. Ullman
ATTN: David P. Flood

Gulton Industries, Inc.
Engineered Magnetics Division
ATTN: Eng. Magnetics Div.

Harris Corporation
Harris Semiconductor Division
ATTN: T. L. Clark, M.S. 4040
ATTN: Wayne E. Abare, M.S. 16-111
ATTN: Carl F. Davis, M.S. 17-220

Hazeltine Corporation
ATTN: M. Waite, Tech. Info. Ctr.

Honeywell, Incorporated
Government & Aeronautical Products Division
ATTN: Ronald R. Johnson, A-1622

Honeywell, Incorporated
Aerospace Division
ATTN: Stacey H. Graff, M.S. 725-J
ATTN: James D. Allen, M.S. 775-D
ATTN: Harrison H. Noble, M.S. 725-5A

Honeywell, Incorporated
ATTN: Tech. Lib.

Hughes Aircraft Company
ATTN: Billy W. Campbell, M.S. 6-E-110
ATTN: Kenneth R. Walker, M.S. D-157

Hughes Aircraft Company
Space Systems Division
ATTN: William W. Scott, M.S. A-1080
ATTN: Edward C. Smith, M.S. A-620

IBM Corporation
ATTN: Frank Frankovsky
ATTN: Harry W. Mathers, Dept. M-41

IIT Research Institute
ATTN: Irving N. Mindel

Intelcom Rad Tech
ATTN: R. L. Mertz
ATTN: Leo D. Cotter
ATTN: Eric P. Wenaas
ATTN: MDC

Kaman Sciences Corporation
ATTN: John R. Hoffman
ATTN: Albert P. Bridges
ATTN: Donald H. Bryce
ATTN: W. Foster Rich
ATTN: Walter E. Ware

Litton Systems, Inc.
Guidance & Control Systems Division
ATTN: R. W. Maughmer
ATTN: Val J. Ashby, M.S. 67

Lockheed Missiles & Space Co., Inc.
ATTN: George F. Heath, Dept. 81-14
ATTN: Benjamin T. Kimura, Dept. 81-14
ATTN: Hans L. Schneemann, Dept. 81-64

DEPARTMENT OF DEFENSE CONTRACTORS (Continued)

LTV Aerospace Corporation
ATTN: Technical Data Ctr.

Martin Marietta Aerospace
ATTN: Jack M. Ashford, MP-537
ATTN: Mona C. Griffith, Library, MP-30
ATTN: William W. Mras, MP-413

Martin Marietta Corporation
Denver Division
ATTN: J. E. Goodwin, Mail 0452
ATTN: Paul G. Kase, Mail 8203

McDonnell Douglas Corporation
ATTN: Tom Ender
ATTN: Technical Library

McDonnell Douglas Corporation
ATTN: Stanley Schneider
ATTN: Raymond J. DeBattista

McDonnell Douglas Corporation
ATTN: Technical Library, C1-290/36-84

Mission Research Corporation
ATTN: William C. Hart

Mission Research Corporation
ATTN: J. Roger Hill
ATTN: David E. Merewether

The Mitre Corporation
ATTN: M. E. Fitzgerald

National Academy of Sciences
ATTN: National Materials Advisory Board for
R. S. Shane, Nat. Materials Advsy

Northrop Corporation
Electronic Division
ATTN: Boyce T. Ahlport
ATTN: George H. Towner
ATTN: Vincent R. DeMartino

Northrop Corporation
ATTN: Orlie L. Curtis, Jr.
ATTN: James P. Raymond
ATTN: David N. Pocock

Northrop Corporation
Electronic Division
ATTN: Joseph D. Russo

Physics International Company
ATTN: Doc. Con. for John H. Huntington

R & D Associates
ATTN: S. Clay Rogers
ATTN: Leonard Schlessinger

The Rand Corporation
ATTN: Cullen Crain

Raytheon Company
ATTN: Gajanan H. Joshi, Radar Sys. Lab.

Raytheon Company
ATTN: James R. Weckback
ATTN: Harold L. Flescher

DEPARTMENT OF DEFENSE CONTRACTORS (Continued)

RCA Corporation
Government & Commercial Systems
Astro Electronics Division
ATTN: George J. Brucker

RCA Corporation
ATTN: E. Van Keuren, 13-5-2

Rockwell International Corporation
ATTN: James E. Bell, HA-10
ATTN: George C. Messenger, FB-61

Rockwell International Corporation
Electronics Operations
ATTN: Mildred A. Blair
ATTN: Alan A. Langenfeld
ATTN: Dennis Sutherland

Sanders Associates, Inc.
ATTN: Moe L. Aitel, NCA, 1-3236

Science Applications, Inc.
ATTN: Frederick M. Tesche

Science Applications, Inc.
ATTN: William L. Chadsey

Science Applications, Inc.
ATTN: J. Robert Beyster
ATTN: Larry Scott

Science Applications, Inc.
Huntsville Division
ATTN: Noel R. Byrn

Simulation Physics, Inc.
ATTN: John R. Uglum

The Singer Company
ATTN: Irwin Goldman, Eng. Management

Sperry Flight Systems Division
Sperry Rand Corporation
ATTN: D. Andrew Schow

Sperry Rand Corporation
Sperry Division
ATTN: Paul Marraffino

Stanford Research Institute
ATTN: Philip J. Dolan
ATTN: Arthur Lee Whitson

Sundstrand Corporation
ATTN: Curtis B. White

Systron-Donner Corporation
ATTN: Gordon B. Dean

Texas Instruments, Inc.
ATTN: Gary F. Hanson
ATTN: Donald J. Manus, M.S. 72

Texas Tech University
ATTN: Travis L. Simpson

TRW Systems Group
ATTN: John E. Dahnke
ATTN: H. S. Jensen

DEPARTMENT OF DEFENSE CONTRACTORS (Continued)

TRW Systems Group
ATTN: A. M. Liebschutz, R1-1162
ATTN: Richard H. Kingsland, R1-2154
ATTN: A. A. Witteles, R1-1120
ATTN: Aaron H. Narevsky, R1-2144
ATTN: Benjamin Sussholtz
ATTN: Lillian D. Singletary, R1-1070
ATTN: Jerry I. Lubell

United Technologies Corporation
Hamilton Standard Division
ATTN: Raymond G. Giguere

DEPARTMENT OF DEFENSE CONTRACTORS (Continued)

Victor A. J. Van Lint, Consultant
Mission Research Corporation
ATTN: V. A. J. Van Lint

Westinghouse Electric Corporation
ATTN: Henry P. Kalapaca, M.S. 3525